

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



OpenStack Cluster HA
Principles and Architecture

OpenStack

高可用集群

(上册)

原理与架构

山金孝 著

国内OpenStack领域一线技术专家在IBM和招商银行等大型企业多年项目经验总结，多位云计算技术专家联袂推荐

立足于生产环境，从原理、架构、部署、运维4个维度为构建高可用OpenStack集群提供完整解决方案



机械工业出版社
China Machine Press

这是一部从原理、架构、部署、运维4个方面系统、深入讲解如何构建高可用OpenStack集群的著作，在理论和实践两个维度为构建高可用OpenStack集群提供了完整的解决方案。

本书从OpenStack终端用户的角色出发，以面向生产系统的OpenStack高可用集群建设为主线，对OpenStack高可用集群的原理和架构进行了深入的剖析，对部署和运维OpenStack高可用集群所依赖的各个技术栈和核心组件进行了详细的讲解。此外，书中还对Ceph和Docker等技术与OpenStack的结合应用进行了详细讲解，尤其是对Kolla项目的介绍，是本书的一大技术特色。

本书分为上下两册：

上册（第1~10章）从理论的角度剖析了OpenStack高可用集群的原理与架构。

架构篇（第1~2章）：介绍了通用云计算参考架构的设计、传统IT架构的高可用设计、云环境下的高可用设计，以及OpenStack高可用集群的架构设计。

原理篇（第3~10章）：首先详细讲解了实现OpenStack高可用集群所必须的集群资源管理器、负载均衡器、消息队列、缓存系统和数据库等OpenStack生态圈的基础技术和高可用软件；其次讲解了OpenStack的计算、网络和存储三大核心组件，以及Ceph的架构设计和使用配置。

下册（第11~15章）从实战的角度讲解了OpenStack高可用集群的部署与运维。

部署篇（第11~12章）：讲解了OpenStack基础架构软件 and 核心组件的高可用部署与实现。全面讲解了OpenStack高可用集群的落地实施过程，并将OpenStack高可用集群的部署进行了代码自动化实现，代码具有稳定的可重现性。

运维篇（第13~14章）：总结了OpenStack高可用集群运维的实践。详细讲解了基于Pacemaker高可用集群的运维，深入分析了Nova实例的高可用和Neutron网络，以及Ceph集群的运维。

拓展篇（第15章）：介绍了基于Docker的OpenStack容器化部署项目Kolla，通过Kolla实现OpenStack容器化部署。

云计算与虚拟化技术丛书



OpenStack Cluster HA
Principles and Architecture

OpenStack 高可用集群

(上册)

原理与架构

山金孝 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

OpenStack 高可用集群 (上册): 原理与架构 / 山金孝著. —北京: 机械工业出版社, 2017.8
(云计算与虚拟化技术丛书)

ISBN 978-7-111-57570-2

I. O… II. 山… III. 计算机网络 IV. TP393

中国版本图书馆 CIP 数据核字 (2017) 第 183664 号

OpenStack 高可用集群 (上册): 原理与架构

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 李 艺

责任校对: 李秋荣

印 刷: 北京诚信伟业印刷有限公司

版 次: 2017 年 9 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 37.75

书 号: ISBN 978-7-111-57570-2

定 价: 99.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

Foreword 序1

开源项目 OpenStack 经过 7 年的发展，其社区规模日益扩大，使用用户日益增多，在云计算领域和市场的影响力也是不断增长。OpenStack 进入我国不算太晚，但由于语言、时差与开源思维等因素的影响，初期在华人用户和社区的发展稍逊于西方国家，特别是落后于美国。随着 OpenStack 日益成熟，以及我国积极倡导开源精神和自主创新精神，OpenStack 在这几年慢慢出现“西退东进”的趋势。即在西方激烈而残酷的行业竞争后，对 OpenStack 的投资有着慢慢趋缓甚至减退的迹象，而在我国，却有越来越多企业加入到 OpenStack 这个大家庭里，有越来越多的行业和大型企业选择并应用 OpenStack，也有越来越多的中国开发人员为 OpenStack 社区贡献代码，为它的成熟与发展发挥了突出的作用。

随着国内 OpenStack 市场的蓬勃发展，关于介绍 OpenStack 的书籍层出不穷。有些主要是国外经典著作的翻译版，以消除国内 OpenStack 爱好者在语言方面的障碍，但更多是来自那些经过多年探索 and 研究的国内 OpenStack 先行者们的原创作品。他们总结自己的学习成果，分享实践领域的经验教训。本书作者山金孝正是众多先行者中的一员，他参考了诸多社区公开的中英文资料，学习 OpenStack 峰会视频和讨论文档，借鉴专家们的技术分享，并结合自己工作的实践，终于著成本书。

我很荣幸在本书即将出版之际，拿到了一份电子简稿，并通读了一遍。它与其他书籍的不同之处在于，所有技术的阐述都是围绕高可用性展开的。OpenStack 的部署实际上包含了很多软件的使用，涉及面广，除了自身几个关键服务之外，它还涉及很多其他不在 OpenStack 项目里的软件和组件。如何把这些软件使用起来，配置好各种参数，使它们配合工作，从各方各面形成完整的高可用 OpenStack 部署方案，是本书的核心，也是对市场上现有各类 OpenStack 书籍的重要补充。相信读者和 OpenStack 爱好者通过本书的学习，了解 OpenStack 原生的高可用性基础架构，熟悉 OpenStack 各相关领域开源技术和软件的应用，熟练掌握高可用 OpenStack 生产项目实施和运维。

——王庆 OpenStack 基金会个人独立董事 /
英特尔开源技术中心云计算和网络部研发经理

序2 *Foreword*

作为开源云计算的事实标准，OpenStack 得到了诸多厂商和企业用户的支持，越来越多的传统 IT 企业和云创公司正在将 OpenStack 作为企业发展的战略方向。全世界范围内的通信、科研、金融和电力等大中型企业用户都在成为 OpenStack 的超级用户，与此同时，对于计划进行云计算建设的企业而言，OpenStack 也是不可或缺的首要参考选项。纵观云计算这些年的发展史，在开源云计算这场 IT 技术革新的硝烟中，OpenStack 已成为最后的赢家，并且正在主导私有云领域的发展方向。

对于大多数 OpenStack 用户，尤其是中小企业用户，OpenStack 组件复杂、理论知识广泛、部署运维门槛较高和不完善的原生高可用建设支持都是阻碍 OpenStack 在企业用户，特别是传统企业用户中落地部署的主要原因。而对于像金融、证券等企业用户而言，面向生产系统的 OpenStack 集群必须具备服务的高可用性，同时要能够满足企业 7×24 的业务连续性需求，因此建设具备高可用性的 OpenStack 集群，是企业选用 OpenStack 部署云计算必须跨过的技术门槛。

本书从 OpenStack 终端用户的角色出发，以面向生产系统的 OpenStack 高可用集群建设为主线，对部署 OpenStack 高可用集群所依赖的各个基础技术栈和 OpenStack 核心组件进行了详细的原理讲解，并以实战部署的形式演示了 OpenStack 高可用集群部署的过程。此外，书中还对 Ceph 和 Docker 等技术与 OpenStack 的结合应用进行了详细讲解，尤其是 Kolla 项目的介绍，都是本书的一大技术特色。目前国内仍然缺乏一本讲解 OpenStack 高可用部署的书籍，本书填补了这一空白，相信也是很多苦于寻求 OpenStack 高可用建设方案用户的福音。

——陈沙克 九州云信息科技有限公司副总裁

Foreword 序3

在受邀为金孝兄的这本书作序时，我是怀着无比激动和忐忑的心情的：一方面为曾经共同在 IBM 战斗过的兄弟能够在开源与云计算环境中有如此专业的经验分享而激动，另一方面其实担心同样作为工程师、架构师的自己不善言语，无从下笔。但当读罢此书时，却也真心觉得想要写些什么，来感慨此书的一些出众之处。

“一切皆服务”是云计算的最终奥义，我一直这样认为。云计算从来都不是一个简单的技术命题，而是一种“让非专业人员也能够自如消费 IT 资源”的手段，是一种资源分配模式的变革。就如同社会从自然经济形态进化到商品经济形态一样，通过云计算，消费者不用再关注于从底层 IT 基础环境到顶层业务实现的全 IT 栈，而是仅需要在云生态环境中选取所需的各类 Infrastructure、Platform、Software 服务来实现自身的业务目标即可。云计算的出现将原本自产自销、纷繁多样的 IT 资源经过标准化、服务化成一种供用户购买的“商品”，从而极大地带动了依托于 IT 资源的各类新兴产业爆发式的发展，诸如互联网+、O2O、物联网等。

这是一本有明确目标的书。我阅读过很多关于云计算、云平台及 OpenStack 的相关书籍，有讲战略的，有讲业务的，有讲技术的，有讲开发的，但大多数都是为了云而云，为了技术而技术。本书却并没有开篇就与读者讨论高可用的模式、集群的实现手段等，而是从“企业为何要进行云计算建设”入手，带着明确的需求与目标来讨论“高可用”在云计算生产环境中的重要地位，从而引出需要考虑的 HADR 关键点，逐步带入实现高可用的技术手段及其原理和实战，使得读者在通读本书的过程中能够带着明确的目标进行研读，提高了读者阅读的兴趣。

这是一本站在巨人肩膀上的书。本书具备严密的逻辑性，这与作者多年来在 IBM 作为一线技术工作者的经历和经验是分不开的。从书中的许多地方都能够透出 IT 界黄埔军校——IBM 的影子，从开篇明义的 CCRA 方法论，到对 HADR 的理解与实现，均站在 IBM 这个蓝色巨人的肩膀上依托于 IBM 的架构思维来审视，加之融合了作者许多成熟经验的实战讲解，使得这本书在整个阅读的过程中能够给读者带来非常连贯、完整和丝丝入扣

的阅读体验。

这是一本实用性卓越的工具书。从事 OpenStack 相关工作的人都很清楚，OpenStack 是一个以功能为目标的项目群，旨在实现越来越全面的云平台功能，而把其本身作为企业生产环境的各非功能目标部分交由用户自行实现，其中最重要的就是高可用部分。而现实情况是，除了 Google、Amazon 这些公有云巨头外，大多数想要使用 OpenStack 作为私有云或混合云的中小企业并不具备全面的规划设计能力，从而无法顺利的将 OpenStack 应用于企业环境。本书恰恰填补了这一空白，从入门的高可用知识，到各类高可用工具的介绍，随后通过高可用集群的部署实践，指导读者一步步完成 OpenStack 高可用的构建，对中小企业将 OpenStack 生产化、商用化起到非常巨大的帮助。

如果你只是想学习 OpenStack 到底为何物，能干什么，我不推荐这本书，你可以上社区进行充分学习，比任何人著作的任何书籍都有帮助；但如果你想将 OpenStack 切实地应用起来，而不是纸上谈兵、简单玩玩而已，那我非常推荐这本书，它能够极大地减少你在浩瀚互联网海洋中摸爬滚打的时间，有的放矢、逻辑严密地指导你一步一步将一个社区化、功能化的 OpenStack 平台，构建成稳定的、高可用的企业级生产环境 OpenStack 云平台。

——行俊楠 云计算资深架构师

Foreword 序4

随着 IT 技术的不断演进以及商业模式的变革式发展，我们今天面对的技术和商业行为越来越体现出开放、敏捷与共享的特征，而云计算恰恰是这其中的基石。依托于云计算给企业带来的不同于传统 IT 架构的更加开放、灵活与共享的能力，企业可以更加敏捷地组织、实现和运营业务，从而更快速、更有效地面对现今的商业竞争。徜徉云计算的海洋中，伴随着诸多 IT 巨头以及不断涌现的创业公司的加入和推广，围绕 OpenStack 的生态圈越发蓬勃，同时也不断扩展到不同的行业、企业和业务场景中。OpenStack 已经成为开源领域云计算的事实标准，并不断推动开源云计算的发展。

与此同时，随着 OpenStack 开源分享精神与企业行业应用的深入结合，如何更加合理地将开源技术与企业的个性化场景结合？如何使得社区共建的开源能力能够满足企业级服务的效率要求？尤其是如何向企业级应用场景提供高可用的解决方案？这些都是开源精神与技术真正能够走入企业级市场必须面对的挑战。在过去面对诸多的企业级需求场景中，我曾经不断面临这些挑战，也深深地感受到在浩瀚的开源技术与信息中，要能够发掘出满足不同业务场景、体现不同个性化需求、解决独特架构体系下的技术难点，是多么的复杂与艰辛！面对这爆发式的信息浪潮，人们对技术的渴求已转为对所获取技术的质量和效率诉求，大量未经验证和无法确认的交流、分享在互联网上如瘟疫一般大肆传播于各大论坛、引擎，读者对其验真、确认的过程犹如沙海淘金，收效甚微。

山金孝先生历时经年，在大量的研究中去其糟粕，取其精华，凝其专业经验与开放信息而著此书，让我不禁为之大为赞叹！本书立意明确，步步以实践为核心，深入浅出地覆盖了如何构建企业高可用云计算的重要技术内容。精读之后让我对其匠人之心深感敬佩，也为我后续企业级产品建设与服务提供了非常多的启发与思考！感谢山金孝先生为 OpenStack 的推广及发展的用心与贡献！

——邹恒滨 四川汇揽熙云科技有限公司产品研发总监

前言 Preface

为什么写这本书

OpenStack 在云计算和 IT 基础架构领域的影响力已毋庸置疑，从 OpenStack 社区成立至今，短短几年时间里，其贡献者和用户已遍及全球各个地区和行业，OpenStack 已然成为开源领域云计算的事实标准。近年来，国内 OpenStack 初创企业不断涌现，华人企业和工程师在 OpenStack 社区中的影响力和贡献占比不断上升，以华为、中兴为主的传统 IT 企业在 OpenStack 社区不断提升自己的影响力，并在社区的董事决策中占有一席之地。此外，99Cloud、United Stack、Easy Stack、UMCloud、AWCloud 等 OpenStack 创业公司对 OpenStack 的贡献也处于全球领先的位置，可以说在国内云计算大潮的推动下，OpenStack 正在不断渗透并重构各个行业的 IT 架构。纵观 2016 年，可将其看成是 OpenStack 在国内企业用户中真正落地的元年，以国网、电信、移动和中海油等为主的大型国有企事业单位，以银联、邮蓄、兴业数金和众多地方商业银行为主的金融企业，以上交、西交、东南大学和人民在线、山西农业云、湖北楚天云等为主的科研政企单位，以及上汽集团、复兴医药等工业制造业都在部署或正式上线 OpenStack 私有云，当然还有更早便在使用 OpenStack 的诸多互联网企业。OpenStack 作为开放、包容的基础架构云平台，在国外更有像 AT&T、CERN、PayPal、BMW、eBay 和 Walmart 等重量级的大型用户，以及 IBM、HPE、Dell、Cisco、RedHat、Intel 和 Oracle 等 IT 巨头的参与。从全球目前的云计算环境发展趋势而言，企业借助 OpenStack 构建私有云已成为多数用户云化并重构数据中心的首要选择。

自 2010 年 OpenStack 的 Austin 版本发布以来，历经 7 年的时间，OpenStack 社区已发行了第 15 个版本 Ocata，其稳定性和功能在不断增强。同时，在 Liberty 版本中引入“大帐篷”概念后，OpenStack 以极为开放、包容的姿态不断整合已有或新生的 IT 技术，尤其是对 Ceph 存储和 Docker 容器技术的成功整合，真正显示出其在云计算大潮下中流砥柱的地位和集成引擎的强大一面。尽管在这几年的发展过程中也饱受指责、批评和诟病，同时还受到 Docker 后来居上甚至取而代之的威胁，但是 OpenStack 依然一路高歌向前，并以拥抱一切竞争对手的姿态不断完善和巩固自己的主导地位。时至今日，可以毫不夸张地说，在

云计算领域，没有 OpenStack 实现不了的功能，也没有 OpenStack 整合不了的技术，或者说在开源领域，OpenStack 已成为了云计算的代名词。

OpenStack 虽然如此火爆，但是想要实现面向生产环境的 OpenStack 高可用云计算环境却并非易事，尤其对于高度依赖传统 IT 架构的企业而言，服务高可用一直是企业部署和使用 OpenStack 难以跨越的鸿沟，而这也成为了 OpenStack 真正落地并走向普通企业用户需要解决的“最后一公里”。OpenStack 以开源共享的方式为用户走向云计算提供了便捷之道，但是其内部组件之复杂、涉及技术栈之多、版本更新之快以及部署维护之困难往往也超出很多用户的预期。围绕这些问题，OpenStack 社区孵化了很多新项目以期解决被诟病最多的入门困难、部署困难和维护困难等问题。但是为了实现 OpenStack 的自动化部署，这些项目无一不又重新引入了新的理论技术，并从其他维度增加了用户使用 OpenStack 的学习和维护成本，而且这些项目并没有很好地解决企业用户最为关心的 OpenStack 服务高可用问题。以上种种，归其原因，在于社区曾明确 OpenStack 的高可用应交由用户的基础架构软件而非上层 OpenStack 项目来实现。因此，对于 OpenStack 企业用户而言，在 OpenStack 社区上游功能交付和终端用户部署实施之间一直横亘着一个难题，即如何在使用 OpenStack 的过程中通过集成高可用基础架构软件，实现保证业务连续性的 OpenStack 高可用集群。

虽然目前各个 OpenStack 厂商都有自己的 OpenStack 高可用解决方案，但是采用厂商定制商业方案似乎有违使用 OpenStack 的初衷，而且市面上的 OpenStack 书籍多以 OpenStack 理论讲解和功能部署为主，却没有一本专门面向 OpenStack 高可用实施部署的中文书籍（虽然在 OpenStack 官方网站和其他互联网站点中也能找到 OpenStack 高可用建设的相关资料，但是这些资料或者藏头露尾，或者零散不全，且以英文资料居多）。秉承共享精神，为了给国内社区用户，尤其是新入门的 OpenStack 用户提供一套完整的 OpenStack 高可用部署参考方案，同时对 OpenStack 进行由底层基础架构软件至上层 OpenStack 核心组件的原理分析与高可用部署的一站式讲解，我们决定编写一本关于 OpenStack 高可用集群原理、部署与运维的书籍，对以 OpenStack 为核心的技术栈进行全面剖析，即从理论知识准备到高可用实战操作，再到后期的高可用集群运维进行透彻的分析和介绍，同时围绕 OpenStack 生态圈，对 Ceph 和 Docker 与 OpenStack 的集成应用进行实战讲解。希望本书能够为企业用户在 OpenStack 的应用部署中提供微薄之力。

本书的主要内容和特色

本书分为上下两册，理论与实战结合，全面讲解了 OpenStack 的技术知识点。上册讲解了 OpenStack 相关的基础架构软件，如集群管理软件 Pacemaker、负载均衡及高可用软件 HAProxy 和 Keepalived、缓存系统 Memcached 和 Redis、数据库 MariaDB 和 MongoDB 以及消息队列系统 RabbitMQ 等基础软件的理论知识，也可以了解到 OpenStack 三大核心组件——计算（Nova）、存储（Cinder/Ceph）和网络（Neutron）的架构原理及使用方式。

下册从实战角度讲解了如何对 OpenStack 的基础架构软件 and 核心组件项目进行高可用集群部署，然后介绍了如何在实际应用中 对 OpenStack 高可用集群进行运维分析与故障解决。

整体而言，本书从项目实施的角度，按照“理论基础—实战部署—后期运维”的方式进行循序渐进的讲解，围绕 OpenStack 生态圈，不仅介绍了 OpenStack 的组件项目，还对其依赖的基础架构软件进行了完整介绍，同时对当前较为热门的 Ceph 和 Docker 在 OpenStack 中的集成应用也进行了分析和实战演示。

本书面向的读者

书中以 Linux 系统运维为基础，通过 Pacemaker 集群软件为各项服务提供高可用性，涉及负载均衡、数据库、缓存系统、消息队列和云计算领域的存储、网络以及计算资源虚拟化等诸多知识点，是一本整合了 OpenStack 生态圈的技术书籍，非常适合 OpenStack 入门初学者、运维工程师和 Open-Stack 高可用架构工程师及 Ceph 存储管理员等从业人员阅读。

此外，本书也适合高校在读本科生或研究生用作 OpenStack 研究、部署和实践的参考书。通过对本书的学习，读者将可以从 OpenStack 入门级别走上高可用 OpenStack 生产项目实施和运维工程师的台阶。

如何阅读本书

阅读本书之前，读者应该具备一定的 KVM 虚拟化知识、SAN 网络或分布式存储知识以及 Linux 系统运维、集群管理和高可用相关方面的基础知识。由于篇幅较多，本书分为上、下两册，按照：架构篇（第 1~2 章）、原理篇（第 3~10 章）、部署篇（第 11~12 章）、运维篇（第 13~14 章）以及扩展篇（第 15 章）的结构进行编排，读者如果仅关注于某个阶段的参考学习，可直接参考目录结构进入相关章节。按章节顺序，本书讲述了如下内容。

第 1 章描述了云计算建设的必要性，同时对用户在公有云与私有云之间的决策提供建设性的参考，并对企业实施云计算的进阶路线提供指引和参考架构，此外还对传统 IT 架构和云计算环境下的高可用架构设计进行了介绍。

第 2 章对 OpenStack 高可用架构的功能组件和集群核心服务项目进行了介绍，还对 Redhat 和 Marientis 两大 OpenStack 领导厂商的高可用架构进行了详细介绍，并与其他 OpenStack 厂商的高可用架构进行了对比分析。

第 3 章介绍了 OpenStack 高可用架构中的集群资源管理器 Pacemaker 的原理、架构和使用方法。

第4章介绍了 OpenStack 高可用架构中的集群负载均衡与高可用软件 HAProxy 和 Keepalived 的原理、架构和使用方法。

第5章介绍了 OpenStack 高可用集群中的消息队列系统 RabbitMQ 的原理、架构和高可用配置与使用方法。

第6章介绍了 OpenStack 高可用集群中的缓存系统 Memcached 和 Redis 的原理及使用方法。

第7章介绍了 OpenStack 高可用集群中的关系型数据库 MariaDB 和非关系型数据库 MongoDB，同时对这两种数据库的高可用配置与使用方法进行了详细介绍。

第8章介绍了 OpenStack 高可用集群中核心服务之一的计算服务 Nova 项目，对 Nova 的架构原理、使用配置以及服务高可用均进行了详细介绍。

第9章介绍了 OpenStack 高可用集群中核心服务之一的网络服务 Neutron 项目，对 Neutron 的插件式架构、高可用配置以及不同的 Neutron 网络模式及其使用方法进行了详细介绍。

第10章介绍了 OpenStack 高可用集群中核心服务之一的块存储服务 Cinder 和 Ceph 分布式存储集群，对 Cinder 块存储的架构、配置和多后端存储及其使用进行了详细介绍，同时对 Ceph 在 OpenStack 中的集成使用进行了实战演示。

第11章介绍了 OpenStack 基础架构组件的高可用部署，以实战部署的方式讲解了如何通过 Pacemaker 集群部署高可用的 OpenStack 基础服务组件。

第12章介绍了 OpenStack 核心服务组件的高可用部署，以实战部署的方式讲解了如何通过 Pacemaker 集群部署高可用的 OpenStack 核心服务组件。

第13章介绍了如何运维基于 Pacemaker 的 OpenStack 高可用集群，并对 Neutron 网络和 Nova 计算服务的高可用性进行了详细分析。

第14章介绍了 Ceph 分布式存储集群的运行维护经验，并提供了如何配置和自定义使用 Ceph 集群的参考。

第15章介绍了如何通过 Docker 容器与 OpenStack 集成项目 Kolla 进行 OpenStack 的容器化部署，同时对 OpenStack 的 Kolla 项目进行了详细介绍。

勘误和资源

在本书的写作过程中，笔者参考了很多 OpenStack 官方社区的资料和历届 OpenStack 峰会的讨论文档与视频，同时也参考了很多开源软件的官方资料和技术专家的经验分享，诚恳希望能为 OpenStack 爱好者与从业者呈现一本涵盖基础理论与高可用实战部署和运维的参考书籍。但是由于 OpenStack 社区版本变化之快，使得任何 OpenStack 相关书籍都难以跟上每年两次的 OpenStack 发行版本，加之笔者水平有限，书中难免存在技术延后和谬误观点，若书中有任何不妥之处，恳请读者批评指正。读者可将意见发送至邮箱

ynwssjx@126.com 或者通过微信公众号“OpenStackGeek”进行反馈，我们将实时跟进 OpenStack 社区的发展变化，并吸取读者的宝贵意见。另外，本书涉及的 OpenStack 高可用集群部署源代码已全部上传至 GitHub，读者可以从网站 <https://github.com/ynwssjx/OpenStack-HA-Deployment> 下载查看并参考实现。

致谢

开源共享是人类历史上最伟大的精神之一，在此向 Linux 创始人和开源精神领袖 Linus Torvalds 致敬，向 OpenStack 项目发起者 NASA 和 Rackspace 致敬，向 OpenStack 社区所有参与者和无私的代码贡献者致敬。

本书的编写历时半年有余，在工作和生活极为繁忙的阶段，笔者仍然坚持每日查阅资料 and 整理文章，期间得到了招商银行很多同事和领导的关心，同时也得到了很多 IBM 前同事和领导的支持，在此一并谢过。正是你们的关心和支持，才使得我在繁忙的工作之余仍然怀着一颗敬畏之心进行写作。

在本书的策划和写作期间，机械工业出版社华章分社的杨福川先生和李艺女士给予了极大的关心和帮助，在此感谢杨福川先生对本书的策划和李艺女士对全文的审阅校对，正是你们的辛勤付出才有了本书的问世。

另外，还要感谢我的妻子杨彩凤女士在写作期间对我生活上的照顾与理解，在多少个深夜与凌晨，正是你的理解与支持让我能全身心地投入写作中。在此也要感谢我的父母，感谢你们的默默养育和辛勤付出。在本书的写作期间，奶奶的仙逝是我最大的悲恸，谨以此书慰藉奶奶的在天之灵，您一世的慈祥我将永远铭记。

最后，感谢所有为本书的编写提供了帮助、支持与鼓励的朋友们，感谢陈沙克、刘世民、吴业亮等无私分享博文的技术爱好者，感谢所有为 OpenStack 社区无私奉献的企业和志愿者。相信在开源精神的共鸣下，OpenStack 一定会变得更好！

Contents 目 录

序 1
序 2
序 3
序 4
前言

架构篇

第 1 章 云计算架构设计及业务系统 高可用..... 2
1.1 企业为何要进行云计算建设..... 2
1.1.1 政策导向与 IT 发展的必然..... 2
1.1.2 业务导向与 IT 弹性需求..... 4
1.1.3 技术导向与 IT 自动化..... 4
1.1.4 成本导向与 TCO..... 6
1.2 企业如何决策公有云与私有云..... 8
1.2.1 云计算部署模式对比..... 8
1.2.2 如何决策私有云与公有云..... 10
1.3 云计算架构设计与进阶路线..... 13
1.3.1 云计算生态模型..... 13
1.3.2 云计算架构基本模型..... 15
1.3.3 通用云计算参考架构..... 16

1.3.4 云计算实施进阶路线..... 20
1.4 业务系统高可用性概述..... 22
1.4.1 业务系统高可用性..... 23
1.4.2 业务系统容灾恢复..... 24
1.5 传统 IT 架构高可用设计..... 26
1.5.1 传统数据中心 HADR 设计原则..... 26
1.5.2 故障划分与 HADR 高可用实现..... 27
1.6 云环境下的高可用设计..... 29
1.6.1 云计算 HADR 架构设计原则..... 30
1.6.2 云计算 HADR 架构设计实现..... 33
1.7 本章小结..... 36

第 2 章 OpenStack 高可用集群 架构概述..... 37
2.1 OpenStack 高可用集群功能组件..... 37
2.1.1 集群控制节点..... 38
2.1.2 集群计算节点..... 39
2.1.3 集群存储节点..... 40
2.1.4 集群网络节点..... 41
2.1.5 集群负载均衡器..... 43
2.1.6 集群网络拓扑..... 44
2.2 OpenStack 高可用集群服务组件..... 47
2.2.1 认证服务 Keystone..... 47

2.2.2	镜像服务 Glance	50
2.2.3	计算服务 Nova	52
2.2.4	块存储服务 Cinder	54
2.2.5	网络服务 Neutron	57
2.2.6	控制面板 Horizon	59
2.2.7	其他 OpenStack 服务	60
2.3	Redhat OpenStack 高可用部署 架构	63
2.3.1	Redhat OpenStack 高可用集群 部署架构	63
2.3.2	Redhat OpenStack 高可用集群 服务规划	67
2.4	Mirantis OpenStack 高可用部署 架构	71
2.4.1	Mirantis OpenStack 高可用集群 部署架构	72
2.4.2	Mirantis OpenStack 自定义高可 用集群架构	76
2.5	其他厂商 OpenStack 高可用部署 架构介绍及对比分析	79
2.5.1	Juniper Networks OpenStack 高可用部署方案	80
2.5.2	HPE OpenStack 高可用部署方案	81
2.5.3	TCP Cloud OpenStack 高可用 部署方案	83
2.5.4	Paypal OpenStack 高可用部署 方案	84
2.5.5	Oracle OpenStack 高可用部署 方案	87
2.5.6	OpenStack 高可用部署方案对比 分析	87
2.6	本章小结	89

原理篇

第3章 集群资源管理系统

3.1	Pacemaker 概述	93
3.2	Pacemaker 集群分类	95
3.3	Pacemaker 集群架构	97
3.4	Pacemaker 内部组件	98
3.5	Pacemaker 集群配置信息管理	99
3.5.1	Pacemaker 集群状态信息	100
3.5.2	Pacemaker 集群配置信息	101
3.6	Pacemaker 集群管理工具 PCS	108
3.6.1	PCS 命令行工具	108
3.6.2	PCS 用户接口界面	110
3.7	Pacemaker 集群资源管理	113
3.7.1	集群资源代理	113
3.7.2	集群资源约束	118
3.7.3	集群资源类型	120
3.7.4	集群资源规则	124
3.8	本章小结	126

第4章 集群负载均衡系统

4.1	Keepalived 概述与配置	128
4.1.1	Keepalived 及 LVS 概述	128
4.1.2	Keepalived 工作原理	133
4.1.3	Keepalived 调度算法	136
4.1.4	Keepalived 路由方式	137
4.1.5	Keepalived 配置与使用	138
4.2	HAProxy 概述与配置	144
4.2.1	HAProxy 概述	144
4.2.2	HAProxy 配置	146
4.2.3	HAProxy 监控页面	151
4.2.4	HAProxy 配置参考	154

4.3 本章小结	158	6.2.1 Redis 缓存概述	204
第5章 集群消息队列系统	159	6.2.2 Redis 数据交换	205
5.1 AMQP 概述	160	6.2.3 Redis 数据持久化	206
5.2 RabbitMQ 概述	161	6.2.4 Redis 数据高可用	207
5.3 RabbitMQ 工作原理	167	6.2.5 Redis 高可用配置	209
5.4 RabbitMQ 基本配置	169	6.2.6 Redis 集群概述	216
5.5 RabbitMQ 集群基础	170	6.2.7 Redis 在 OpenStack 中的应用	218
5.5.1 RabbitMQ 集群概述	170	6.3 本章小结	219
5.5.2 RabbitMQ 的集群配置	171	第7章 集群数据库系统	221
5.6 RabbitMQ 集群管理	174	7.1 关系型数据库——MariaDB	221
5.6.1 RabbitMQ 集群节点启停	174	7.1.1 MySQL 概述	221
5.6.2 RabbitMQ 的集群节点移除	175	7.1.2 MariaDB 概述	224
5.7 RabbitMQ 的集群队列镜像	177	7.1.3 MariaDB 安装配置	225
5.8 基于 Pacemaker 的高可用 RabbitMQ 集群	181	7.1.4 MariaDB 高可用方案	233
5.8.1 Active/Passive 模式的 RabbitMQ 集群	181	7.1.5 MariaDB Galera Cluster 概述	236
5.8.2 Active/Active 模式的 RabbitMQ 集群	182	7.1.6 MariaDB Galera Cluster 配置	239
5.9 RabbitMQ 在 OpenStack 中的应用分析	187	7.2 非关系型数据库——MongoDB	249
5.10 本章小结	192	7.2.1 NoSQL 概述	249
第6章 集群缓存系统	193	7.2.2 MongoDB 概述	251
6.1 Memcache 缓存系统	193	7.2.3 MongoDB 安装配置	254
6.1.1 Memcache 缓存概述	193	7.2.4 MongoDB Replica Set 概述	258
6.1.2 Memcache 的工作原理	194	7.2.5 MongoDB Replica Set 部署	260
6.1.3 Memcache 的功能特点	196	7.3 本章小结	265
6.1.4 Memcache 集群概述	197	第8章 OpenStack 计算服务	267
6.1.5 Memcache 集群高可用	201	8.1 OpenStack 项目概述	267
6.2 Redis 缓存系统	204	8.1.1 OpenStack 项目概要	267
		8.1.2 OpenStack 版本发行	268
		8.1.3 OpenStack 组织机构	272
		8.1.4 OpenStack 使用情况	274
		8.1.5 OpenStack 服务项目	276

8.2	Nova 项目概述.....	277	第 9 章 OpenStack 网络服务.....	388
8.2.1	Nova 架构设计.....	277	9.1 Neutron 网络概述.....	388
8.2.2	Nova 功能模块.....	282	9.2 Neutron 网络架构.....	394
8.3	Nova 分区与区域.....	285	9.2.1 Neutron 网络架构概述.....	394
8.3.1	Nova 中的 Region.....	285	9.2.2 Neutron Plugin 与 Agent.....	396
8.3.2	Nova 中的 Cell.....	288	9.2.3 Neutron L3 Service 分析.....	402
8.3.3	Nova 中的 Availability Zone.....	292	9.3 Neutron 网络类型.....	408
8.3.4	Nova 中的 Host Aggregate.....	294	9.3.1 Provider 网络.....	408
8.4	Nova Hypervisor 配置概述.....	297	9.3.2 Self-Service 网络.....	411
8.4.1	虚拟化与 Hypervisor 概述.....	297	9.4 Provider 网络部署与分析.....	415
8.4.2	Nova Hypervisor 归类支持.....	303	9.4.1 Provider 网络基于 OpenvSwitch 实现.....	415
8.4.3	Nova Hypervisor 选取配置.....	308	9.4.2 Provider 网络基于 LinuxBridge 实现.....	424
8.5	Nova 主机策略.....	317	9.4.3 Provider 网络创建与验证.....	429
8.5.1	Nova scheduler 主机过滤.....	317	9.5 Self-Service 网络部署与高可用.....	433
8.5.2	Nova scheduler 主机加权.....	324	9.5.1 Self-Service 网络实现.....	433
8.5.3	Nova scheduler 配置选项.....	329	9.5.2 Self-Service 网络高可用.....	450
8.6	Nova 实例创建.....	333	9.6 L3 HA 高可用方案.....	452
8.6.1	Nova 实例创建流程.....	333	9.6.1 L3 HA 高可用部署实现.....	452
8.6.2	Nova 实例状态变更.....	341	9.6.2 L3 HA 高可用验证与分析.....	459
8.6.3	Nova 实例创建方法.....	347	9.7 DVR 高可用方案.....	470
8.7	Nova 实例迁移.....	354	9.7.1 DVR 高可用部署实现.....	470
8.7.1	Nova 实例 resize/migrate 迁移.....	354	9.7.2 DVR 高可用验证与分析.....	477
8.7.2	Nova 实例 live-migration 迁移.....	365	9.7.3 DVR 与 L3 HA 对比.....	492
8.8	Nova 实例高可用.....	376	9.8 DVR/L3 HA 高可用方案.....	493
8.8.1	Nova 实例高可用概述.....	376	9.8.1 DVR/L3 HA 高可用部署实现.....	493
8.8.2	Nova 实例高可用之 Evacuate/ Rebuild.....	378	9.8.2 DVR/L3HA 高可用验证与分析.....	499
8.8.3	Nova 实例高可用之 Pace- maker_remote.....	382	9.9 本章小结.....	511
8.9	本章小结.....	387	第 10 章 OpenStack 存储服务.....	512
			10.1 OpenStack 存储概述.....	513

10.1.1	OpenStack 存储分类对比	513			
10.1.2	OpenStack 存储后端选择	515			
10.2	Cinder 块存储	519			
10.2.1	Cinder 块存储架构	519			
10.2.2	Cinder 块存储使用	520			
10.2.3	Cinder 块存储插件	524			
10.2.4	Cinder LVM 插件实现	529			
10.2.5	Cinder NFS 插件实现	534			
10.2.6	Cinder Multi-Backends 实现	540			
10.3	Ceph 存储系统	545			
10.3.1	Ceph 背景概述	545			
10.3.2	Ceph 架构设计	547			
10.3.3	Ceph 工作原理	553			
10.3.4	Ceph 部署实现	559			
10.4	Ceph 集成 OpenStack	564			
10.4.1	Ceph 集成 OpenStack 概述	564			
10.4.2	Ceph 集成 OpenStack 准备	566			
10.4.3	Ceph 集成 Glance	569			
10.4.4	Ceph 集成 Cinder	571			
10.4.5	Ceph 集成 Nova	574			
10.4.6	Ceph 集成 OpenStack 验证	578			
10.5	本章小结	581			
	部署篇				
	第 11 章 OpenStack 高可用集群				
	基础服务部署	584			
11.1	OpenStack 集群高可用离线部署	584			
11.1.1	制作 OpenStack 离线安装 pip 源	585			
11.1.2	制作 OpenStack 离线安装 yum 源	592			
11.2	OpenStack 集群高可用部署架构 设计	599			
11.2.1	OpenStack 高可用部署实验 环境架构	599			
11.2.2	OpenStack 高可用部署生产 环境架构	603			
11.2.3	OpenStack 高可用部署软件 拓扑架构	608			
11.3	OpenStack 集群高可用部署实验 环境准备	610			
11.3.1	控制节点 VMware 宿主机 准备	611			
11.3.2	控制节点 KVM 虚拟机 准备	617			
11.3.3	计算节点 VMware 虚拟机 准备	624			
11.4	OpenStack 高可用集群基础 服务部署	625			
11.4.1	Pacemaker 集群管理软件 部署	625			
11.4.2	HAProxy 负载均衡器高可用 部署	628			
11.4.3	MariaDB 关系数据库高可用 部署	633			
11.4.4	Memcache 缓存系统高可用 部署	639			
11.4.5	RabbitMQ 消息队列高可用 部署	640			
11.4.6	MongoDB 非关系数据库 高可用部署	643			
11.5	本章小结	646			

第 12 章 OpenStack 高可用集群

核心服务部署 647

12.1 OpenStack 控制节点服务高可用

部署 647

12.1.1 Keystone 认证服务高可用

部署 648

12.1.2 Glance 镜像服务高可用

部署 655

12.1.3 Cinder 块存储服务高可用

部署 660

12.1.4 Neutron 网络服务高可用

部署 665

12.1.5 Nova API 服务高可用部署 676

12.1.6 Ceilometer 数据采集服务

高可用部署 682

12.1.7 Heat 编排服务高可用部署 687

12.1.8 Horizon 控制面板服务

高可用部署 691

12.2 OpenStack 计算节点服务高可用

部署 694

12.2.1 OpenStack 计算节点高可用

实现概述 694

12.2.2 OpenStack 计算节点高可用

方案分析 695

12.2.3 OpenStack 计算节点 Pace-

maker 高可用集群分析 696

12.2.4 OpenStack 计算节点 Pace-

maker 高可用集群实现 697

12.3 OpenStack 集群服务高可用

验证 707

12.3.1 OpenStack 高可用集群

功能性验证 707

12.3.2 OpenStack 高可用集群

高可用验证 722

12.4 本章小结 731

运维篇

第 13 章 OpenStack 高可用集群

运维最佳实践 734

13.1 Pacemaker OCF 资源代理故障

诊断分析 735

13.1.1 Pacemaker 集群 OCF 资源

代理使用介绍 735

13.1.2 Pacemaker 集群 OCF 资源

代理定义语法 737

13.1.3 Pacemaker 集群 OCF 资源

代理调试诊断 744

13.2 Pacemaker 集群调试与管理维护 749

13.2.1 Pacemaker 集群日志系统

设置 749

13.2.2 Pacemaker 集群日志构成

分析 751

13.2.3 Pacemaker 集群日志调试

分析 755

13.2.4 Pacemaker 集群 GUI 管理

界面 758

13.3 OpenStack 实例高可用原理分析

与问题诊断 765

13.3.1 OpenStack 高可用集群计算

节点资源配置 765

13.3.2 OpenStack 高可用集群

Fence_compute 分析 766

13.3.3 OpenStack 高可用集群

NovaEvacuate 分析	771
13.3.4 计算节点高可用实现原理与 问题诊断分析	774
13.4 OpenStack Neutron 网络理解与 故障问题诊断	781
13.4.1 OpenStack Neutron 网络概念 基础	781
13.4.2 OpenStack Neutron 网络深入 理解	784
13.4.3 OpenStack Neutron 网络故障 分析	803
13.5 OpenStack 日常管理与运维	811
13.5.1 OpenStack 日志设置管理与 使用	811
13.5.2 OpenStack 故障实例数据 检查恢复	813
13.5.3 OpenStack 故障计算节点 实例恢复	816
13.5.4 OpenStack 实例间浮动 IP 地址管理	818
13.5.5 OpenStack 服务运行缓慢 解决方案	819
13.5.6 OpenStack 配置文件及数据库 备份	821
13.6 本章小结	824

第 14 章 Ceph 存储集群运维最佳 实践

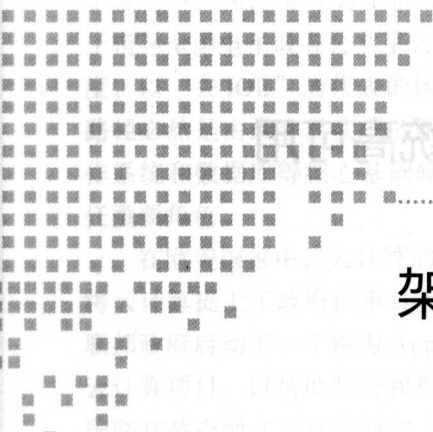
14.1 Ceph 规划配置与性能调优	825
14.1.1 Ceph 硬件配置推荐	825
14.1.2 Ceph 配置文件设置	829

14.1.3 Ceph CRUSH 自定义	843
14.1.4 Ceph SSD 应用场景	854
14.1.5 Ceph 性能调优关键	862
14.2 Ceph 运维与常见故障处理	867
14.2.1 Ceph OSD 与 PG 状态	867
14.2.2 Ceph OSD 节点增删	871
14.2.3 Ceph MON 节点增删	875
14.2.4 Ceph Journal 故障维护	877
14.2.5 Ceph OSD 故障硬盘更换	880
14.2.6 Ceph 存储节点停机维护	881
14.2.7 Ceph 容量耗尽解决方案	883
14.2.8 Ceph 常用命令使用参考	886
14.3 本章小结	891

扩展篇

第 15 章 Docker 容器部署 Open- Stack

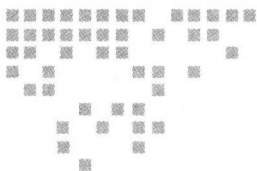
15.1 OpenStack 与 Docker	894
15.1.1 容器与虚拟机的现状	894
15.1.2 OpenStack 融合 Docker	897
15.2 Kolla 项目介绍	900
15.2.1 Kolla 项目使命	900
15.2.2 Kolla 及其现状	905
15.2.3 Kolla 内部组件	907
15.3 Kolla 容器化部署 OpenStack	915
15.3.1 系统部署环境准备	915
15.3.2 制作 Docker 镜像	917
15.3.3 部署 Docker 容器	919
15.3.4 OpenStack 功能验证	920
15.4 本章小结	924



架构篇

■ 第1章 云计算架构设计及业务系统高可用

■ 第2章 OpenStack 高可用集群架构概述



Chapter 1 第 1 章

云计算架构设计及业务系统高可用

云计算架构

近些年云计算少有的降温，恰从事实上证明云计算已经脱离争论不休和布道宣扬的早期雏形，并真正落地行走，步入每个人的生活。当前阶段，越来越多的企业投入到云计算大潮中，不管是公有云建设还是私有云建设，都掀起了一股云计算建设热潮。但在云计算部署实施的过程中，我们需要的不再是是否上云，而是如何上云，如何部署实施，如何在云上满足企业自己的业务高可用需求。本章将从云计算架构设计讲起，介绍云计算建设的架构模型以及云计算建设的进阶路线，同时还会介绍企业业务系统的高可用以及如何设计云上业务系统的高可用架构，从而为企业建设云计算提供架构设计和业务可用性设计的参考。

1.1 企业为何要进行云计算建设

1.1.1 政策导向与 IT 发展的必然

时至今日，云计算早已成了 IT 领域的“常态”。从 2006 年 Google 公司开始提出“云计算”这一 IT 概念以来，云计算的发展不仅得到了各商业机构和科研院校的大力支持，也得到了世界各国政府在云计算发展和成熟过程中的极大推动。以我国为例，2010 年 10 月国务院发布《关于加快培育发展战略性新兴产业的意见》，将云计算纳入战略性新兴产业；同月发改委发布《关于做好云计算服务创新发展试点示范工作的通知》，确定北京、上海、杭州、深圳以及无锡五城市先行开展云计算服务创新发展试点示范工作；2011 年国务院发布《关于加快发展高技术服务业的指导意见》，将云计算列入重点推进的高技术服务业；2012 年财政部国库司发布《政府采购品目分类目录（试用）》，增加了 C0207“运营服务”，包括：软件运营服务、平台运营服务、基础设施运营服务三类，分别对应云服务中的 SaaS、PaaS

和 IaaS 服务, 国家关于云计算的政策逐渐从战略方向的把握走向推进实质性应用, 政府及公共管理部门采购云计算服务的重要制度障碍开始被逐步打破; 同年 7 月, 《“十二五”国家战略性新兴产业发展规划》出台, 将云计算工程作为中国“十二五”发展的二十项重点工程之一; 2013 年 3 月工信部发布了《基于云计算的电子政务公共平台顶层设计指南》, 用于指导全国电子政务公共平台的建设^①。近年来, 中国政府将信息安全上升到国家安全的高度, 以“去 IOE”为代表的国产软硬件对进口软硬件的替代趋势是不可逆转的。而替代的路径必然是由最容易实现的服务器等硬件设备到云计算和云存储等软件与服务, 最终到操作系统和数据库等核心基础软件平台上, 因此, 云计算在我国信息安全的建设过程中将担任重要角色。

在欧美国家中, 云计算的起步相对较早。作为云计算的起源地, 美国政府在 2009 年便将云计算提上了政府议事日程, 同年, 奥巴马政府宣布了一项长期性的云计算政策, 随后联邦政府启动了一个称为 Apps.gov 的网站, 通过这个网站展示和提供得到联邦政府认可的云计算项目, 以帮助政府和相关机构更好地理解并接受云计算的理念^②。从 2010 年起, 美国联邦政府便在云计算研究上增加预算, 同时美国政府推行的云计算计划也涉及了政府的各个领域。欧洲国家虽然在云计算领域也属于跟随者, 但是其在云计算使用方面已经相当成熟, 如最为熟悉的欧洲粒子物理研究机构 CERN 以及 IBM 与荷兰政府合建的都柏林云计算中心等。

而在亚洲国家中, 除了中国政府大力支持云计算以外, 日本政府也非常重视云计算的研究和应用, 日本的 OSS (Open Source Software) 推进联盟还专门成立了云计算战略研究小组, 主要研究云计算发展, 并领导开展相关云计算项目。同时, 云计算在日本的汽车、医疗、电力及影视行业都得到了广泛使用。同处亚洲的韩国则在 2008 年举办了“The Clouds 2008”大会, 开始了云计算基础设施建设的举措, 2009 年, 韩国政府决定向云计算领域投入巨资以提升本国云计算的全球占有比例^③。

任何事物的出现和发展在其背后都有必然之因, 云计算也不例外。尼古拉斯·卡尔在其专著《IT 不再重要: 互联网大转换的制高点——云计算》一书中以历史类比的方式论证了云计算的必然性: 正如爱迪生发明了电灯泡, 随后企业建设了自有电厂, 而后出现了现在的中心化电厂和电力传输网络, 计算资源也一样, 如果人类历史仍然还在向前发展, 那么云计算必然在 IT 领域出现。此外, 加州大学伯克利分校的《伯克利云计算白皮书》则从具体的云计算实现和运作原理以及其自助服务等方面佐证了卡尔的云计算必然性论证^④。因此, 拥抱云计算, 着手建设和使用云计算, 这是历史大势的必然, 也是每一位 IT 从业人士新的机遇。

① 工业和信息化部电信研究院. 云计算白皮书 [J/OL]. 2014. <http://www.docin.com/p-1290647881.html>.
http://www.cena.com.cn/2015-05/18/content_276657.htm.

② 杨振东. 基于云计算的中小企业信息化建设模式研究 [D]. 山东: 中国海洋大学, 2010.

③ 邱群业. 企业私有云计算基础架构研究与设计 [D]. 广东: 华南理工大学, 2012.

④ 孙定. 云计算必然性的经典论证 [J]. 石油工业计算机应用: 2011, 1:17-17.

1.1.2 业务导向与 IT 弹性需求

在传统 IT 架构中,任何业务系统在上线前都需要进行所需 IT 资源的预估和架构设计,并且这种预估通常是对未来几年内所需 IT 资源的估算,如果对于前期 IT 资源的投入预计不足,则业务系统性能必然受到影响,最终将会影响到客户的体验和满意度,而如果预估过高,则又会出现 IT 资源闲置和浪费,IT 成本过高的情况。此外,随着企业业务的发展和组织的壮大,前期的 IT 规划势必不能满足不断增加的业务负载,IT 资源的扩展或者架构的重构成为企业必然选择,而传统 IT 架构的扩容方式分为两种:Scale-out 和 Scale-up,也称为水平扩展和垂直扩展。Scale-up 扩展不需要过多考虑应用系统是否具备分布式属性,其主旨是使原有计算能力变得更强大,而不是新增独立的 IT 设备资源并把应用负载分散到其上,因此其实现方式通常就是增加服务器 CPU、内存、硬盘驱动等设备,或者直接购买更强大的服务器替换原设备。而不管是硬件升级还是新旧系统架构的替换,Scale-up 在阶段性扩容时候都会是巨大的成本开支(图 1-1 中“巨额成本支出”),或者在大容量新 IT 架构实施替换完成之前,由于 IT 资源不能及时跟进,企业业务拓展规划只能放弃(图 1-1 中“商机已过”区域)。相比而言,Scale-out 扩展方式与 Scale-up 完全不同,在 Scale-out 中,通常初始投入的 IT 基础设施成本较低,但是 IT 架构可以且易于水平扩展,很多商业 IT 架构尤其是大规模的 Web 应用都遵循这一架构模式,即将应用负载分散到集群节点中,然后整合数据集并采用面向服务的架构设计模式。Scale-out 在扩展时将采取小规模递增的水平扩展形式,尽管这种扩展形式相比 Scale-up 要高效和经济很多,但是在扩展时候依然要对未来一段时间的 IT 需求进行预测,而这种预测通常会导致容量过剩,即通常所说的“烧钱”,而且,由于是小规模递增,还需保持持续的人工检测以免出现 IT 资源供不应求的紧急情况。此外,Scale-up 和 Scale-out 两种形式的扩展都需要对 IT 基础设施进行一定的前期成本投入,同时两种扩展方式从本质上来说都是在 IT 资源不够情况下的被动扩展^①。

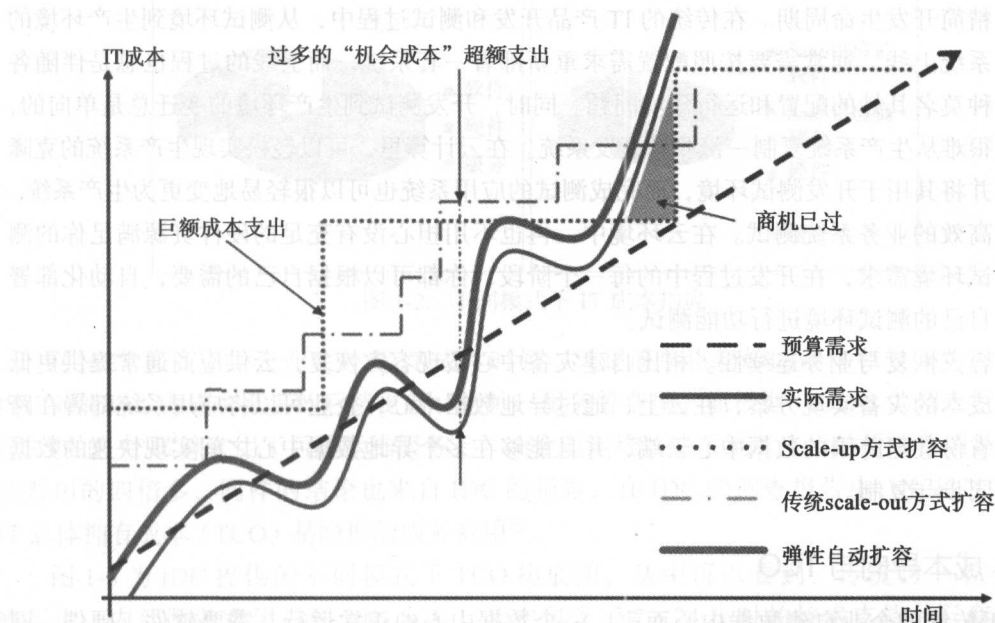
在云计算中,按需和弹性是其固有特性,也是最为突出的特性,通过计算能力的按需弹性扩展,使得企业可以随时调节 IT 基础设施资源,以使其长期逼近不断扩展或收缩的企业实际 IT 资源需求(图 1-1 中波形实线),当企业能够以业务为导向,不断根据实际业务系统的 IT 需求来弹性动态地调节 IT 资源时,Scale-up 和 Scale-out 扩展方式下的 IT 资源预测过高和预测不足所带来的成本浪费和 IT 不足等难题,都将得到非常理想的解决。

1.1.3 技术导向与 IT 自动化

对于任何单位和组织而言,业务和人员的发展壮大必然伴随着 IT 系统的庞大和复杂化,最明显的变化便是数据中心的设备数量与能耗的增加。然而,从成本控制的角度来看,企业的决策者总是希望以不变的技术人员成本来支撑不断增长的业务需求所带来的 IT 扩容,在传统过分依赖手工运维的 IT 架构中,这似乎是不可实现的悖论,而在云计算时代,

① Eugene Gorelik. Cloud Computing Models [D]. Massachusetts: Massachusetts Institute of Technology, 2013.
<https://aws.amazon.com/cn/whitepapers/architecting-for-the-aws-cloud-best-practices/>.

这却是数据中心的常态。云计算的出现，将使你的数据中心变得更加自动化，业务系统将会变得更加高可用，云计算所带来的技术优势体现如下^①：



- ❑ 基础设施部署脚本自动化。在云计算时代，我们不用再考虑部署一套系统在设备的搬运、上架、上电、布线和系统安装上需要花费多少时间，也不用再担心系统参数配置问题。在云计算领域 devops 的倡导下，一切皆需自动化，如果你需要部署一套系统，通过可编程的基础设施（通常是 API 驱动的）即可快速重复地创建和部署满足你所需要的系统。整个过程中，复杂的底层基础设施完全被屏蔽，用户只需调用云服务提供者的 API 接口编辑简单的脚本命令即可快速批量部署自己的系统。
- ❑ 紧急需求弹性伸缩。市场需求与商业灵感总是瞬息万变，商业上的任何预定计划都难以与实际市场需求完全吻合，或者市场膨胀，需要 IT 应用随之扩展以满足业务处理需求，或者商机已逝，要求 IT 应用随之消减以满足成本控制需求。在云上，任何意外的 IT 需求都可以实现自动伸缩，整个应用伸缩的过程完全以市场的驱动为向导，实现了 IT 需求随着市场供需的波动而自动伸缩，在一定误差范围内，既满足企业 IT 需求，又不浪费企业 IT 成本。
- ❑ 预期需求主动伸缩。在企业全部的 IT 系统中，有很大一部分 IT 系统主要用于内部办公与流程处理，对于内部使用系统，IT 资源的预期需求相对稳定，在做出合适的 IT 需求和预算之后，通过云计算，便可以较低的成本和较快的速度来响应你的 IT

① <https://aws.amazon.com/cn/whitepapers/architecting-for-the-aws-cloud-best-practices/>.

规划。如果在后期需要对 IT 需求进行增加或者剥减,云计算仍然能够实时满足你 IT 需求的伸缩。

- ❑ 精简开发生命周期。在传统的 IT 产品开发和测试过程中,从测试环境到生产环境的系统上线,通常需要按照配置需求重新部署一套系统,而上线的过程也总是伴随各种莫名其妙的配置和运行环境问题。同时,开发测试到生产环境的变迁总是单向的,很难从生产系统复制一份测试开发系统。在云计算里,可以轻松实现生产系统的克隆并将其用于开发测试环境,而完成测试的应用系统也可以很轻易地变更为生产系统。
- ❑ 高效的业务系统测试。在云环境中,再也不用担心没有充足的硬件资源满足你的测试环境需求,在开发过程中的每一个阶段,你都可以根据自己的需要,自动化部署自己的测试环境进行功能测试。
- ❑ 容灾恢复与业务连续性。相比自建灾备中心实现容灾恢复,云供应商通常提供更低成本的灾备实现方案,在云上,通过异地数据中心,企业可以将应用系统部署在跨省份甚至跨国的数据中心云端,并且能够在多个异地数据中心之间实现快速的数据同步与复制。

1.1.4 成本导向与 TCO

对于传统的企业自建数据中心而言,一个数据中心的正常运行,需要软件、硬件、网络基础设施、监控和测试工具、安全产品以及空调电力等基础设施的支持和大量运维人员的投入,而这些都是企业资本开支(Capital Expenditure)的一部分。而在云计算时代,企业 IT 设备及专业管理人员的长期大笔资本成本(CaPEX)被转化为以按需租赁为形式的运营成本(OpEx),在云计算模式下评估信息系统的资源投资回报率(ROI)时,不能再采用传统 IT 模式下的评估方式,以 SaaS 服务为例,如果按照传统的购买软件许可方式来进行 ROI 的评估,则云计算模式下的 SaaS 费用可能要比购买软件许可费用贵,因为企业很少会考虑定期租用软件导致的企业运营成本变化。因而,在云计算环境下,对 IT 生命周期的全部可见成本和隐形成本衡量应该采用更为科学的 TCO(Total Cost of Ownership)评价方式。TCO 是由著名的 IT 领域 Gartner 公司提出的 IT 全生命周期成本评价方式,下面我们以云计算 SaaS 为主,通过 TCO 的方式来重点分析传统 IT 模式和云计算模式下企业 IT 成本的支出对比。

在传统模式下,企业一次性的 IT 投资成本包括硬件和软件,在整个生命周期内还包括后续服务的相关费用;而云计算模式下,企业是通过支付租赁费用形式来使用信息系统的,企业拥有信息系统的使用权而没有所有权。在传统的软件许可证模式下,企业需要支付大量的硬件、软件以及后续的专业服务费用,在计算 IT 投资的过程中,IT 预算一般包括硬件、软件和软件服务费,但是电力和空间成本却很少被预算考虑进去,然而这种持续支出的成本却占据了传统 IT 总成本的很大部分;而云计算模式下,主要成本和费用则是需要长期计算能力的软硬件租赁费和服务费,以及自建信息中心的少量硬件和软件费用。两种模式下 IT 成本构成如图 1-2 所示。

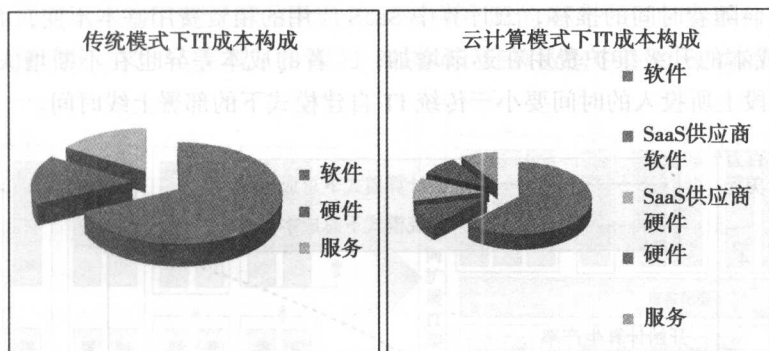


图 1-2 不同模式下 IT 成本构成

传统 IT 模式下，企业在计算 IT 成本时候，只是简单将初始的硬件和软件费用计算在内，而忽略了占成本更多的后续更新、维护、服务等费用。而根据 Gartner 的调查报告，超过 75% 的 IT 预算被用于后续运维服务上，而企业运维这些系统所支出的后续费用是购买系统费用的四倍多，同样的结论也来自 IDC 的报告，在 IDC 的调查报告，有超过 70% 的 IT 总体拥有成本 (TCO) 是隐形的服务费用^①。

图 1-3 为 IDC 提供的不同模式下 TCO 构成图，从中可以看到，传统 IT 一次性预算购买中，内部服务费用占到 70% 左右，其中包含了 7×24 小时的技术支持，厂商服务台 (Help Desk)，硬件 / 软件获取与维护等费用。

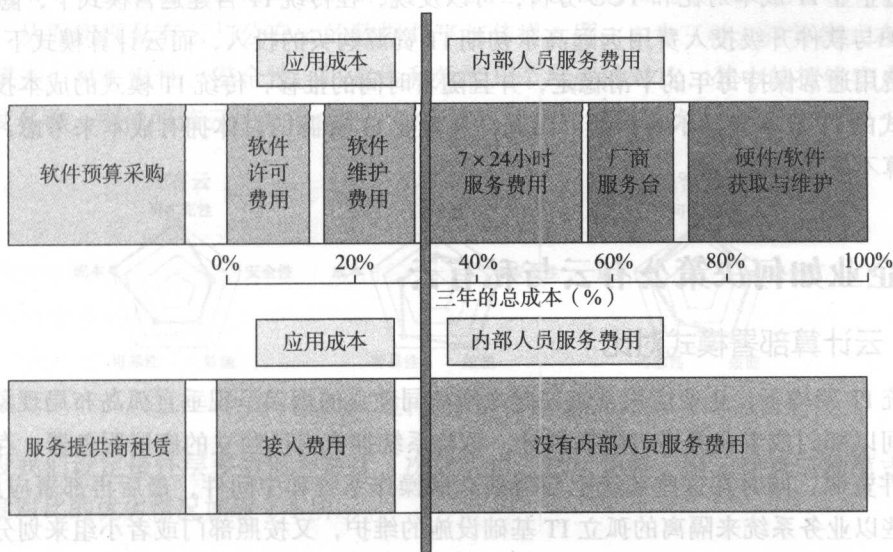


图 1-3 总体拥有成本 (TCO) 构成

图 1-4 为 IDC 对传统的 IT 内部建设和云计算中应用层 SaaS 的成本差异所做的比较，

① 杨振东. 基于云计算的中小企业信息化建设模式研究 [D]. 山东: 中国海洋大学, 2010.

从中可以看出,随着时间的推移,云计算中 SaaS 应用的租赁费用基本不变,而传统模式的内部 IT 建设成本的升级维护费用在逐渐增加,二者的成本差异也在不断增大,同时 SaaS 在初始生产阶段上所投入的时间要小于传统 IT 自建模式下的部署上线时间。

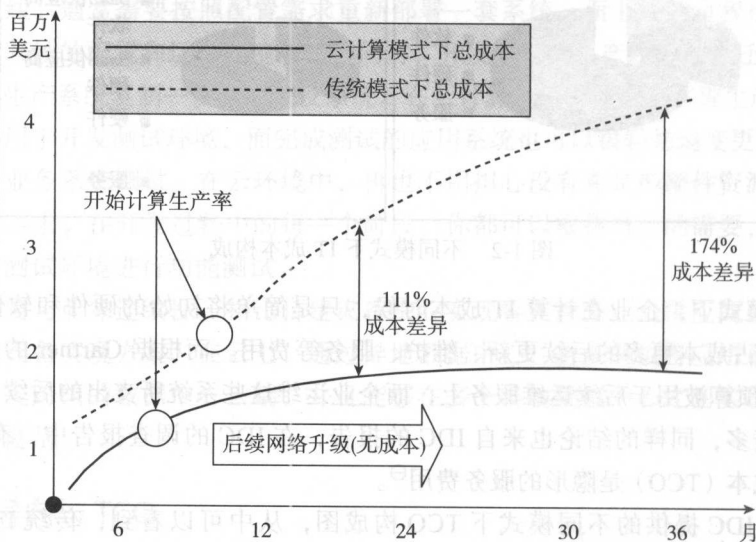


图 1-4 IDC 对传统软件与 SaaS 的 TCO 对比图

通过企业 IT 成本对比和 TCO 分析,可以发现,在传统 IT 自建运营模式下,隐形的 IT 后续维护与软件升级投入费用远远高于初期 IT 资源购买的投入,而云计算模式下 IT 资源的投入费用通常保持每年的平滑稳定,并且随着时间的推移,传统 IT 模式的成本投入与云计算模式的 IT 成本差异不断扩大。因此,从企业 IT 资源的总体拥有成本来考虑,企业拥抱云计算不失为明智之举。

1.2 企业如何决策公有云与私有云

1.2.1 云计算部署模式对比

传统 IT 架构下,企业应用系统彼此之间如同独立的烟囱呈现垂直孤岛布局现象,各个系统之间以部门或者业务进行隔离划分,每套系统拥有自己独立的底层服务器、存储、网络等硬件资源,同时在这些系统上部署独立的操作系统和中间件,最后再部署应用程序,而对这些以业务系统来隔离的孤立 IT 基础设施的维护,又按照部门或者小组来划分,即不仅 IT 资源呈现独立孤岛现象,IT 部门人力资源也呈现零散状态,同时还需分设相应的硬件与软件岗位,这种孤岛现象最大的特点便是 IT 资源和人力资源不能充分利用,出现资源的极大浪费。与传统 IT 不同,云计算采用 IT 资源池化的概念,将全部硬件基础设施进行虚拟化之后,组成统一的大资源池,通过 Hypervisor 虚拟化层分配资源池中的资源供上层应

用系统使用,同时云平台管理员通过软件定义硬件资源的形式管理配置各种硬件资源,实现开发运维的自动化。图 1-5 呈现了传统 IT 架构向云计算数据中心 IT 架构的演化。

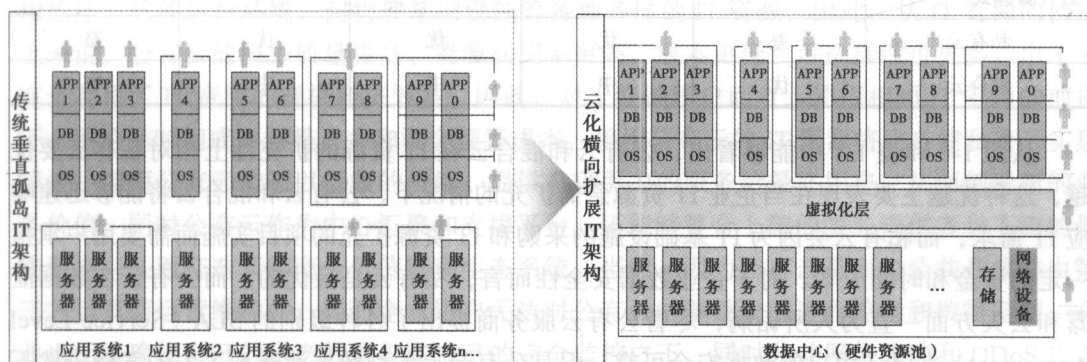


图 1-5 传统 IT 架构向云数据中心架构的演变

通常而言,云计算部署模式分为公有云、私有云和混合云,传统的划分里面也包含社区云,但是其极少被提及和部署。私有云主要构建于企业或者组织机构内部,企业可以选择开源或者商业的云计算软件进行私有云的建设,例如开源的 OpenStack 和 CloudStack 等,或者商业的 VMware vCloud Director 等。公有云是由第三方提供的计算资源集合,公有云的使用和访问通常需要基于 Internet,用户以租赁按需付费的形式购买第三方提供的计算、存储、网络等服务。混合云是二者的集合,即企业自建私有云与公有云能够进行数据通信和互访,从而实现私有云与公有云的数据共享与传递。图 1-6 为三种云计算模式在企业 IT 资源的成本、可扩充性、安全性、可靠性和效能几个维度的对比,其中的折线实点越接近考评维度名称,则说明该云计算模式下,对应实点的当前维度越优。

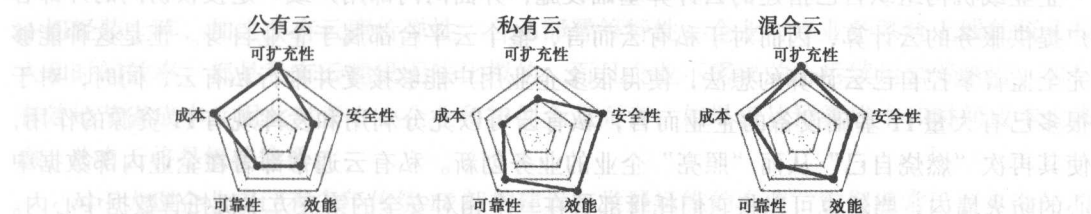


图 1-6 三种云计算模式对比

假设我们设定最外层多边形为最优,次之为中,然后为差,则云计算三种模式下各个不同维度对比的优劣情况如表 1-1 所示。

表 1-1 云计算三种模式不同维度对比

对比维度 云计算模式	可扩充性	安全性	可靠性	效能	成本
公有云	优	差	中	差	优

(续)

对比维度 云计算模式	可扩充性	安全性	可靠性	效能	成本
私有云	差	优	优	优	差
混合云	优	中	中	中	中

从图 1-6 和表 1-1 中能够看出，公有云和混合云在 IT 资源的扩充性上相对私有云要优越，这种优越主要表现在当企业 IT 资源急需扩充的情况下，公有云和混合云将能够迅速响应 IT 需求，而私有云会因为 IT 基础设施的采购和 IT 资源扩充的项目实施而需要用户承担一定的资金和时间成本。对于企业数据安全性而言，私有云是最优的，而公有云在数据泄露和丢失方面一直为人所诟病，尽管公有云服务商提出了各种级别的 SLA（Service Level Agreement）来保证用户的数据安全可靠，但是公有云上各种数据泄露和丢失的案例仍然不绝于耳。此外，从成本维护上来考虑，公有云由于具有 IT 资源的规模经济优势，因此使用公有云无疑是成本最优的，而私有云和混合云均需要企业出资出力来自建，同时还需后期的运维服务的费用，所以公有云在云计算投入的成本上要低于私有云和混合云，当然这也是公有云对私有云的最大优势之一。

1.2.2 如何决策私有云与公有云

私有云是为某个特定客户的使用而单独构建的，因而客户可以对自身数据、安全性和服务质量具有最有效的控制权。建设私有云的企业拥有对自己基础设施的全部控制，而并非如公有云一样只有租赁后的使用权却没有所有权，并且企业可以在此基础设施上按照自己的需求进行各种定制开发，并在此基础设施上部署自己的应用程序。由于私有云是由单一企业或机构组织自己搭建的云计算基础设施，并面向内部用户或一定授权访问的外部客户提供服务的云计算，因而对于私有云而言，整个云平台都属于企业自身。正是这种能够完全监督掌控自己云计算的想法，使得很多企业用户能够接受并推行私有云，同时，对于很多已有大量 IT 基础设备的企业而言，私有云可以充分利用和发挥现有 IT 资源的作用，使其再次“燃烧自己”从而“照亮”企业的业务创新。私有云通常部署在企业内部数据中心的防火墙内，当然也可以将它们托管部署在一个相对安全的第三方主机托管数据中心内。在内部 IT 人力资源充分的情况下，私有云可由公司自己内部的 IT 部门进行建设，当然如果自身技术实力不足，也可跟云提供商合作并由第三方来进行私有云的建设。在很多类似“专用式托管”的私有云建设模式中，很多云计算提供商可以对 IT 基础设施进行安装、配置和运营维护，因而从基础设施层面上给予一个企业数据中心专用云的全面支持，在这种模式下，公司对于云资源的使用具有与自己内部自建的私有云同等程度的控制能力，同时将基础设施的运行维护转交由更为专业的第三方公司负责，因而企业 IT 部门能够更专注于 IT 资源的使用和业务系统的开发测试等工作。

公有云通常指第三方 IT 公共资源供应商为用户提供的 IT 资源服务，用户利用自己内

部的 IT 设施通过 Internet 接入云端的公有云来进行 IT 资源的按需付费使用。因为公有云是由专门的云服务提供商来运营的, 并通过运营商自己的基础架构直接向最终用户提供从应用程序、软件运行环境, 到物理基础设施等各种各样的 IT 资源, 因此, 从 IT 资源的使用上来说, 公有云的服务质量更高、资源利用率更高、成本更低、可选择性也更多。由于公有云利用了 IT 资源的规模经济效益, 因此, 对于终端用户而言, 使用相同 IT 资源的前提下, 不论是时间成本还是人力和资金预算成本, 租用公有云的 IT 资源将比自建私有云实惠很多, 因而公有云能够以低廉的价格, 提供有吸引力的服务给最终用户, 从而创造新的业务价值, 同时公有云作为中心汇聚和支撑平台, 还能够整合上游的服务提供者和下游的最终用户, 从而打造新的商业价值链和生态系统。当然, 因为公有云是面向公共开放并由第三方运营商运营维护的, 因而企业用户无法对公有云的基础设施进行控制和按需定制, 企业也无法确保自己的生产数据仅在自己的完全监控之下, 同时公共网络故障和 DDoS 攻击等不确定因素, 也为公有云的使用埋下了数据安全和访问中断的隐患。

关于私有云与公有云的选取决策, 因为每个企业自身发展状况的不同, 并没有固定的评判标准来指引企业选择私有云或者公有云。对于任何一个企业来说, 私有云与公有云的选择应该考虑多个方面, 从而实现企业的最优 IT 发展规划。通常, 企业在正式拥抱云计算之前, 总是需要考虑以下几个方面, 根据自身情况决定选择私有云还是公有云:

(1) 成本预算

就中小企业和初创企业而言, 选择私有云就意味着需要进行一次性的大量投资来采购软、硬件设备, 并且需要投入大量的人力物力进行项目建设与后期的运营维护, 同时由于很多初创企业并无 IT 基础设施建设方面的经验和人才, 建设私有云无疑会是一个成本较高风险较大的选择。此外, 很多初创企业急于业务系统上线, 私有云的建设周期也是其难以承受的时间成本。而如果采用公有云, 由于初创企业并无老旧应用云化迁移的负担, 一切轻装上阵, 加之公有云廉价弹性、自动部署等特性, 企业对于业务系统上线的资金投入和时间效率, 都是私有云建设无法比拟的, 而且企业不需要关心后续运营等事宜, 可以有效地节省成本。因此对于中小企业和初创企业而言, 如果仅从预算成本和时间成本上考虑, 公有云将是极佳选择。

对于大型企业或已有多年传统 IT 架构建设与发展运维的企业, 它们通常已有复杂的 IT 系统架构, 同时 IT 部门也具有一定深度的技术实力, 并有丰富 IT 基础运营经验。如果这类 IT 基础设施已经相当完善的企业果断转向公有云, 则无疑意味着对现有 IT 资产的巨大浪费, 并且要额外在公有云上重新购买计算资源, 并将现有业务系统迁移至公有云上, 这不仅是对资金预算的巨大挥霍, 同时, 现有 IT 系统需要做调整或迁移以适应公有云, 这无疑也要承担巨大成本支出和应用迁移风险, 而选择私有云则只需考虑一次部署成本。因此, 对于已有多年 IT 发展经历和大型企业而言, 选择私有云建设无疑是明智的决策。

(2) 数据安全

在企业自建的私有云环境中, 企业对私有云基础架构到上层 API 接口及其访问控制都

有绝对的控制权，企业完全可以根据自己的规章制度设置私有云的使用访问权限，而且私有云内部的数据流动仅在公司内部防火墙内部，企业数据有着可控较高的安全。但是如果企业采用公有云供应商提供的第三方云服务，就不得不考虑安全和法规等方面的问题。在很多行业里，尤其是很多大型上市公司里，企业除了要遵守国家法律规定外，相应的行业和上市企业所需要遵循的安全规定和标准也是企业所不能忽视的，然而，由于目前国内公有云计算市场很大程度上还处于成长探索阶段，并不是所有的云服务提供商都能够满足企业所要遵循的法规要求，因此企业在转向公有云之前，一定要谨慎评估这些方面。另外，对于使用公有云的企业客户，企业和公有云上系统之间的数据传输是建立在 Internet 连接上的，这种传输方式对企业数据而言，增加了泄露和数据被盗风险。同时，由于公有云集中了大量客户数据，公有云供应商的网络系统往往成为网络恶意攻击的首选，尤其是近些年来流行的 DDoS 攻击往往造成用户访问的大面积中断，除了网络攻击，公有云运营商所遭遇的意外事故，如各种“闪电事件”和“挖机事件”都会直接大面积中断用户访问。因此，从数据安全可靠性来看，私有云将是最佳选择，而公有云在数据安全方面的问题一直是阻碍企业转向公有云的主因。

（3）应用集成

目前多数公有云运营商都能提供 PaaS 服务，例如关系型数据库 RDS 或者对象存储服务 OSS，尽管这些功能强大的 PaaS 服务也能很好地支撑企业的海量数据存储业务，但是对于很多大型企业客户来说，转向公有云不得不花费巨大的财力和精力来更改自己已有的应用架构，才能使用公有云上的 PaaS 平台服务，而且通常会存在一些更糟糕的情况，就是很多特殊行业目前的应用与公有云服务完全无法对接。以阿里云为例，其关系数据库服务 RDS 目前只支持 MySQL 或 SQLServer 数据库的迁移，而并不支持 Oracle 和 DB2 数据库，这就意味着使用 Oracle 和 DB2 的用户必须在迁移之前改变自己的数据库底层设计，需要先将数据迁移至 MySQL 或 SQLServer 数据库，然后再迁移至 RDS，否则无法使用阿里云的 RDS 服务，而在某些行业里，除了传统的 Oracle 和 DB2 数据库，还保留有 Sysbase、Informix 等数据库。因此，对于这些客户而言，迁移至公有云就意味着更改应用接口或者直接架构重建，这样的改动对于企业而言无疑是巨大的成本开支并要承担潜在风险。

相对于公有云集成现有应用系统所带来的巨大风险和成本开支，私有云对于企业的现有应用系统的兼容性要好很多，私有云在建设过程中便可根据企业应用系统的特点进行底层服务的定制，以使其很好地满足企业应用系统的需求。因此，对于企业来说，业务系统迁移至自己的私有云并不需要任何应用层面的更改，对于应用系统的使用者而言，迁移是透明不被感知的。因此，从现有应用系统的集成角度来看，公有云优势不再，私有云才是最佳的选择。

（4）数据可靠

使用公有云，则意味着企业数据必将存放在公有云上。尽管很多公有云运营商声称可以实现 5 个 9 的 SLA，即 99.999% 的数据可靠性，但是用户对于自己数据的控制仍然非常

有限,例如,企业用户无法掌握具体的数据备份和冗余策略,也完全不知备份和冗余数据的存放方式和存储情况。而在私有云上,虽然用户并不一定拥有比公有云更为健壮的冗余和备份策略,但是冗余和灾备策略对于私有云用户而言是公开的,企业可以掌控自己的业务数据、备份数据以及具体的策略,并且可以根据自己业务系统的重要程度和数据量大小来设置业务数据冗余和备份策略,而不是采用公有云供应商提供的统一标准策略。因此,从数据的可靠和可控性来看,私有云仍然具有很大的优势。

(5) 资源体验

公有云下的资源是运营商维护下的资源池,用户对这些资源并不具有绝对的监控和掌控性,企业用户与公有云运营商之间的资源供应关系主要通过签订 SLA 来维护,正如企业通常无法完全使用电信运营商承诺的网络带宽一样,公有云供应商承诺的网络带宽或者存储 IOPS 并不能绝对实现 SLA 的标准,甚至会有比较大的资源折扣,因而带宽和计算能力都无法得到绝对的保证。由于企业对私有云拥有完全控制能力,而且通常只向单一内部客户提供 IT 资源服务,企业也可以根据自身需要进行资源大小的调整以适应自己的应用系统,并且可以完全保证这些资源不被用于其他。因此,从资源的使用体验而言,私有云肯定更能满足各种应用系统的需求。

(6) 扩展能力

弹性扩展是云计算的固有特性,也是云计算真正的价值所在。弹性扩展的主要目的在于使得 IT 资源随着业务系统的需求而弹性伸缩,从而实现既不浪费 IT 资源,又能迅速保证和满足业务系统的 IT 需求。公有云通常拥有私有云无法达到的资源池,因此其资源向上可扩展的空间也是私有云无法实现的,故而公有云在应对强烈变化的业务 IT 需求上要远优于私有云,虽然私有云也具有弹性伸缩的特性,但是其在运行成本和响应迅速膨胀的业务 IT 需求上仍然不如公有云廉价和便捷快速。对于很多初创企业或者业务时好时坏的行业,选择弹性公有云将具有很大优势,而如果采用私有云建设的模式,则可能出现 IT 资源浪费或者跟不上业务的发展。因此,从资源的弹性角度来看,公有云比私有云更有优势,尤其是业务发展较快的初创企业,公有云将是不错的选择。

1.3 云计算架构设计与进阶路线

1.3.1 云计算生态模型

在进行云计算架构设计之前,我们有必要了解清楚云计算生态链由哪些角色和环节构成,从而在我们的云计算架构设计过程中,才能清楚定位哪些模块对应生态链中的哪些角色,最后在云计算项目的实施过程中,我们仍然需要整合或借助云计算生态链中的诸多角色资源才能最终建设自己的云计算平台。目前云计算生态链主要有六大角色:云设备供应商、云系统构建商(云平台开发商、系统集成商)、云应用开发商、云服务运营商(云资源服务提供商、云平台服务提供商、云应用服务提供商)、云服务部署/交付商、云服务销售

商和最终用户。其中，云服务销售商、云服务部署商、云系统构建商、云应用开发商、云软硬设备供应商均是面向云服务运营商提供服务的，而云服务运营商主要面向最终用户提供 IaaS、PaaS、SaaS 服务，云计算生态链各角色之间的关系如图 1-7 所示。

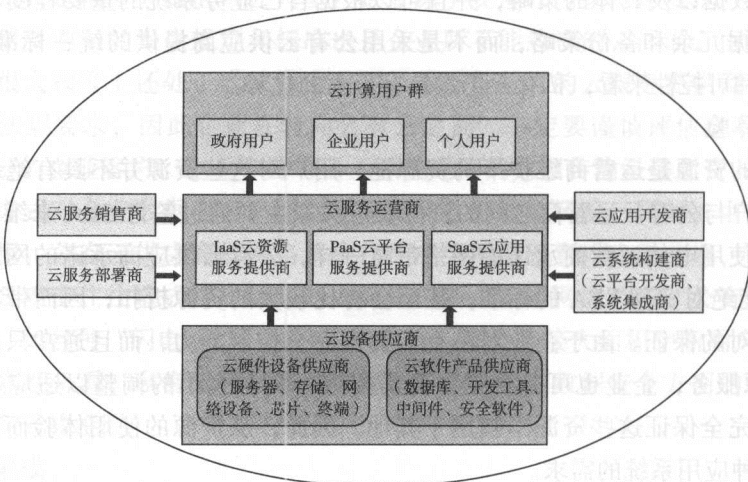


图 1-7 云计算生态链角色关系

云生态链上各个不同的角色分工明确，不同层次的角色需要依赖其他角色提供的服务才能存在。商业伙伴合作机构（Business Partner Consulting, BPC）基于对云计算领域各类企业业务结构的分析与归纳，根据企业面对客户群的不同，为云计算生态链定义了 6 种不同的角色类型，在此基础上根据业务结构的差异细分出了 12 个业务渠道角色，这些不同类型的渠道角色组成了一个标准的云生态系统模型^①，如图 1-8 所示。在 BPC 的云计算生态模型中，渠道角色的层次由上到下递增，下层角色的实现需要依赖上层角色，例如云系统构建商需要云设备提供商提供的软硬件设备，云应用开发商又需要云系统构建商的软硬件集成服务，而云服务运营商在正式提供云服务之前，必须由云应用开发商进行各种行业或通用软件的开发，同时云服务部署商需要使用云运营商的云服务来盈利和进行商业活动。

当然，对于特定的企业而言，不可能全部覆盖云生态链中的每一个角色。根据企业对于自己业务性质的定位，每个企业在进行云计算建设时参与其中的云生态链角色也不尽相同。假如企业根据自身业务特性和 IT 需求决定建设私有云，则云生态链中的云服务运营商和云服务最终用户将合为一体，而如果企业在已有的 IT 基础设施上建设私有云，而不是重新采购新设备，则云设备提供商就不会参与私有云的建设，同时如果企业自身 IT 实力较强，无须云系统集成与云应用开发商，那么企业私有云的建设最终将只有企业自身一个角色存在（扮演几乎全部角色），当然这种情况对于企业 IT 部门的要求是非常高的，所以企业在建设私有云之前，一定要对自身 IT 人员云计算相关的技术水平进行切实估计，通常，在

① <http://www.chinacloud.cn/show.aspx?id=19249&cid=13>

企业 IT 人员较为紧张的情况下引入专业的第三方云系统构建和云应用开发商，不失为明智之策。而如果企业对于云计算的建设定位是使用公有云运营商提供的服务，那么对于企业而言，需要考虑的云生态链角色可能就只剩下云服务运营商和云服务部署商了，企业根据自身 IT 资源的需求，租赁云服务运营商的 IaaS、PaaS、SaaS 服务，如果企业对于公有云的使用不是很熟悉，或者应用系统无法集成到公有云上，则可以引入云服务部署商，通过云服务部署商将自己的应用集成到公有云上，同时对自己的 IT 人员进行云服务培训。

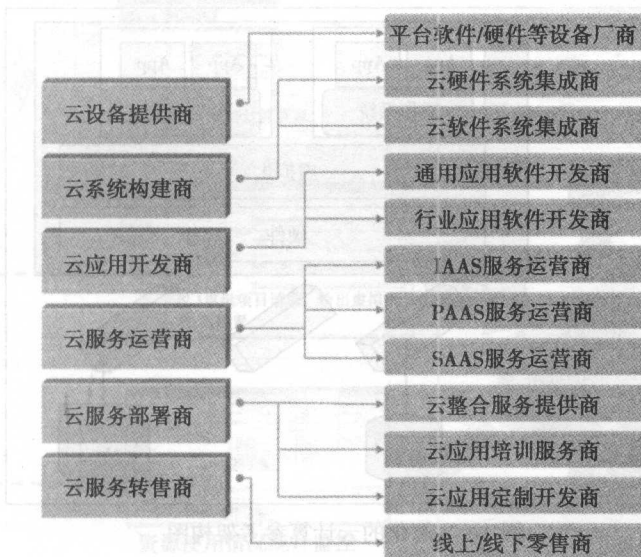


图 1-8 BPC 标准云生态模型

1.3.2 云计算架构基本模型

众所周知，虚拟化是支撑云计算的核心技术和基础，而云计算是面向服务架构（SOA）的一种实现，单纯的虚拟化只是迈向云计算的开始。虚拟化技术的主要目的在于提高资源使用率和能耗效率，并将底层硬件与上层软件进行隔离，使得上层软件及应用的计算需求变得弹性可控，同时极大降低服务器的维护成本开销及风险，提供业务系统的快速容灾恢复和高可用性，伴随虚拟化的出现，业务系统跨物理节点的实时迁移也成为可能。但是虚拟化技术默认并不对外提供抽象的服务组件，一个没有被服务化的虚拟化环境只能称为资源池，并且只有内部管理员才能操作，因此这样的虚拟化环境并没有被云化（Cloud-Enable）。对于任何一个云计算架构来说，建立在虚拟化环境上的抽象服务层都是必须的，因为它隐藏了底层复杂的基础架构设施，并向用户提供功能丰富的云管理接口，只有这样，底层基础设施才能被软件定义，从而实现 IT 资源可编程的灵活控制与访问，图 1-9 为一个简化的云计算架构模型。

在具体的云计算架构实现过程中，每一个厂商或者用户对于抽象服务层的理解都有自己

的特点,但是不管架构如何变化,底层物理基础设施层、中间的虚拟化层、上层抽象服务层的云计算架构框架是不变的,1.3.3 节将会介绍主流通用云计算参考架构,通过分析这些云计算架构,我们将会部署和实施云计算之前,对如何设计云计算架构有一个宏观上的认识。

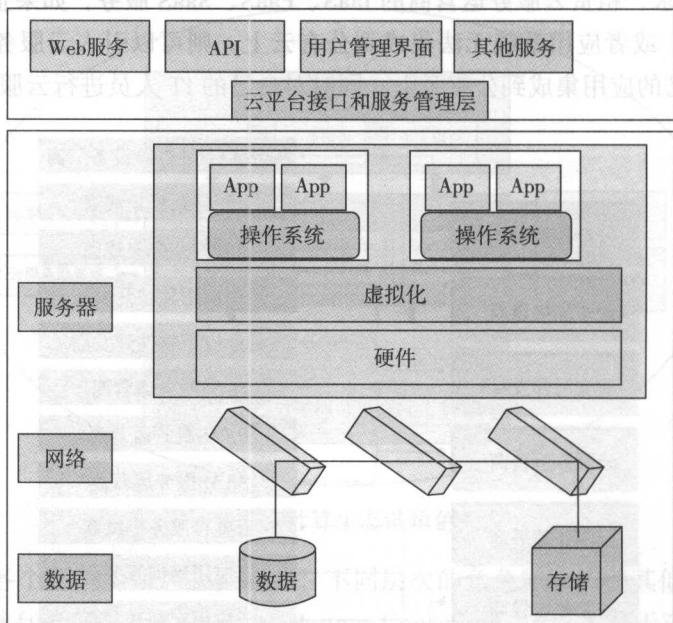


图 1-9 简化的云计算参考架构图

云平台架构的设计最终直接影响到云计算资源的使用,在进一步深化云计算参考架构之前,我们有必要了解在云计算环境下资源的使用和流程应该是怎样的。图 1-10 显示了云环境下企业对 IT 资源的使用管理流程,即简单的申请、审批和使用三部曲,使用者首先需要申请使用云资源,然后云管理员需要视情况审批并创建云资源,最后通知申请者领用云资源,同时企业的 IT 高层应该对本企业的全部 IT 资源使用运行情况进行全局掌控。这是一个简单的云环境下 IT 资源的使用管理流程,也是多数云计算参考架构在设计实现时应该具备的功能,因此,在真正准备和实施云计算之前,应该考虑的一个问题是,如果按照此参考架构功能模块来建设,未来的云环境是否可以实现图 1-10 中描述的基本功能。

1.3.3 通用云计算参考架构

在开始云计算的正式架构与部署实施前,有必要了解通用云计算参考架构的设计模式与组件,然后结合自身实际情况,对通用参考架构的组件进行取舍。参考架构为云计算提供了一个蓝图,蓝图中包括了完整定义的功能模块范围,云计算所要满足的需求条件以及实现云计算的架构决策。云计算参考架构以一种标准化的方法论指导了整个云计算项目从前期规划到后期实施应该参考的每个方面。云计算参考架构也称为 CCRA (Cloud Comput-

ing Reference Architecture), CCRA 定义了构成云计算环境的基本元素, 同时 CCRA 由高层次抽象化的功能模块组成, 每个抽象化的模块均可根据需要进行深化细分^①。图 1-11 是 IBM 提供的云计算参考架构, 也是被业界普遍接受的 CCRA, IBM 的 CCRA 由面向服务的架构 SOA 发展而来, 其在业务支撑模块和运营支撑模块的细化对于很多公有云和私有云建设都具有很好的参考意义。

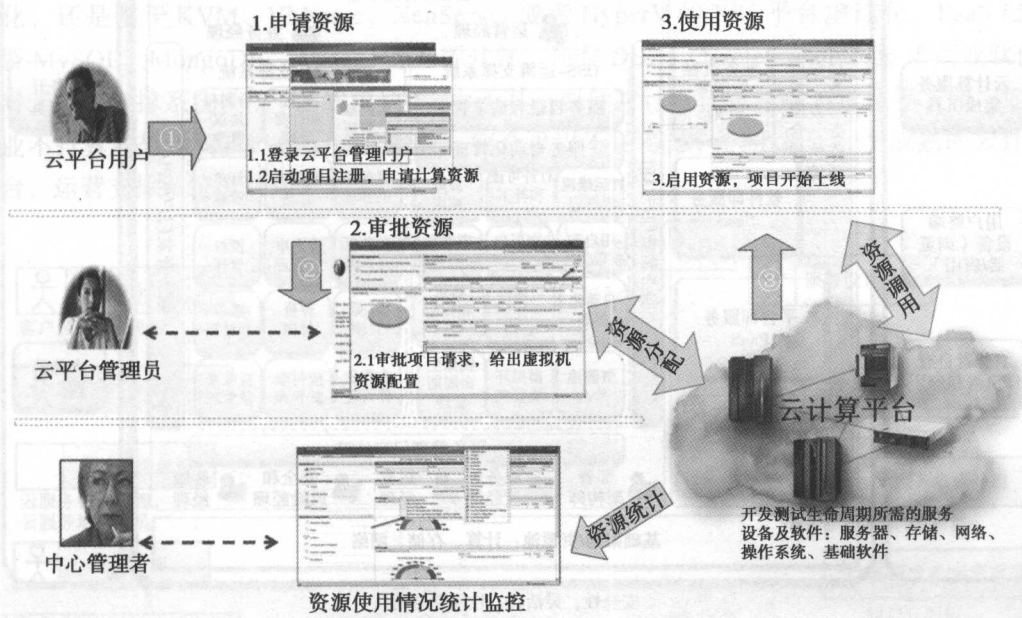


图 1-10 简化的云服务使用管理流程

当然, 要理解图 1-11 中的 CCRA, 需要事先掌握某些角色和术语。从 CCRA 的角色层面来看, IBM 的 CCRA 主要定义了三种角色: 云计算服务消费者、云计算服务运营者和云计算服务开发者。这种角色定义只是对任何云计算场景中均会出现的角色所做出的角色集合定义, 在特定的云计算场景, 这些角色均可再细分为更具体的角色。下面是对这三种抽象云角色的概念解释:

- ❑ 云服务消费者。云服务消费者可以是一个机构组织, 也可以是个人。云服务消费者通过浏览云服务提供者的服务目录来选购自己需要的服务, 同时云服务提供者会对这些呈现给消费者的服务进行计量计费。云消费者通过购买云端的资源服务来满足自己的 IT 资源需求, 同时这种服务可以根据消费者的需求随时终止计费和重新启动计费。
- ❑ 云服务运营者。云服务运营者也称为云服务提供者, 主要负责向云消费者提供云服务。云提供者最核心的部分是通用云管理平台, 只有通过云管理平台才能向消费者提供各种云服务模式, 如 IaaS、PaaS、SaaS 等服务。云管理平台通常分为运营支

① Dr. Menchita, F. Dumlao. Cloud Computing Reference Architecture from Different Vendor's Perspective [J]. International Journal of Emerging Technology and Advanced Engineering: 2013, 3(11): 528-524.

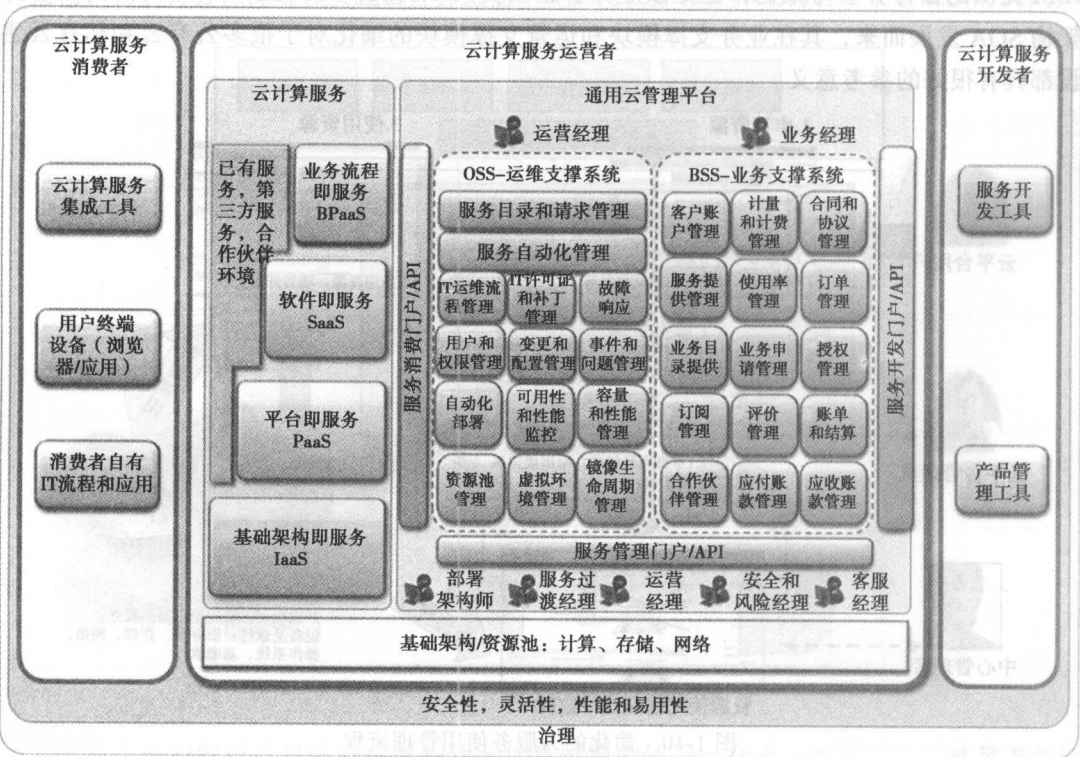


图 1-11 IBM 的云计算参考架 CCRA

❑ 云服务开发者。云服务开发者也是云服务的创建者，虽然其与云服务提供者从角色上进行了独立划分，但是二者也可以同属一个企业组织。云服务创建者主要负责创建可为云消费者使用的云服务，当然这些云服务也必须建立在云运营商提供的云资源上。

CCRA 中的三种角色在具体的云计算场景中将会被细分为多种角色，当然，这些角色也并非存在整个云计算环境生命周期中，某些角色的存在可能只是为了特定需求的满足而临时设置。图 1-12 是对三种云计算角色的细分场景之一，其中云服务提供者又可以细分为很多角色，这些角色囊括了业务层面和运营层面的角色，而云服务消费者和云服务创建者内部也有各种经理角色和管理员角色。

对于多数企业而言，完全按照图 1-11 中的 CCRA 理论模型来进行云计算建设，从技术和成本预算上来看都是难以承受的。更为重要的是，图 1-11 中 CCRA 的很多功能模块对于大多数企业而言都是不必要的。或许对于建设公有云而言，该参考架构更贴近于实际模型，但是，如果企业计划自建私有云，则图 1-11 中云计算参考架构的很多功能模块都可以直接剔除，并且对于私有云的建设而言，云服务消费者、云服务提供者和云服务创建者均为企

业自身（也有可能外包部分工作），而且很有可能一人身兼数种角色，所以，没有任何一家计划云计算建设的企业可以完完全全复制 CCRA 或者其他企业的云计算架构。因此，在标准 CCRA 的基础上，企业应该根据自身已有的 IT 基础设施资源和业务需求进行 CCRA 的定制设计，图 1-13 即是根据图 1-11 进行削减定制的 CCRA，其架构仍然还是仅具代表性的定制架构，企业可以根据自己的 IT 设备情况，选择是进行基于 powerVM 的 IBM 小型机虚拟化，还是基于 KVM、VMware、XenServer 或者 HyperV 的 X86 平台虚拟化，PaaS 层是部署 MySQL、MongoDB、Apache 等开源软件，还是 DB2、Oracle、Weblogic 等商业软件，而对于业务支撑系统模块（虚线模块），由于其主要针对计量计费 and 账单管理等功能，如果企业不打算对外运营自己的云计算服务，完全可以不用考虑，但是对于一个成熟的云计算平台，运营支撑系统模块却是必须的。

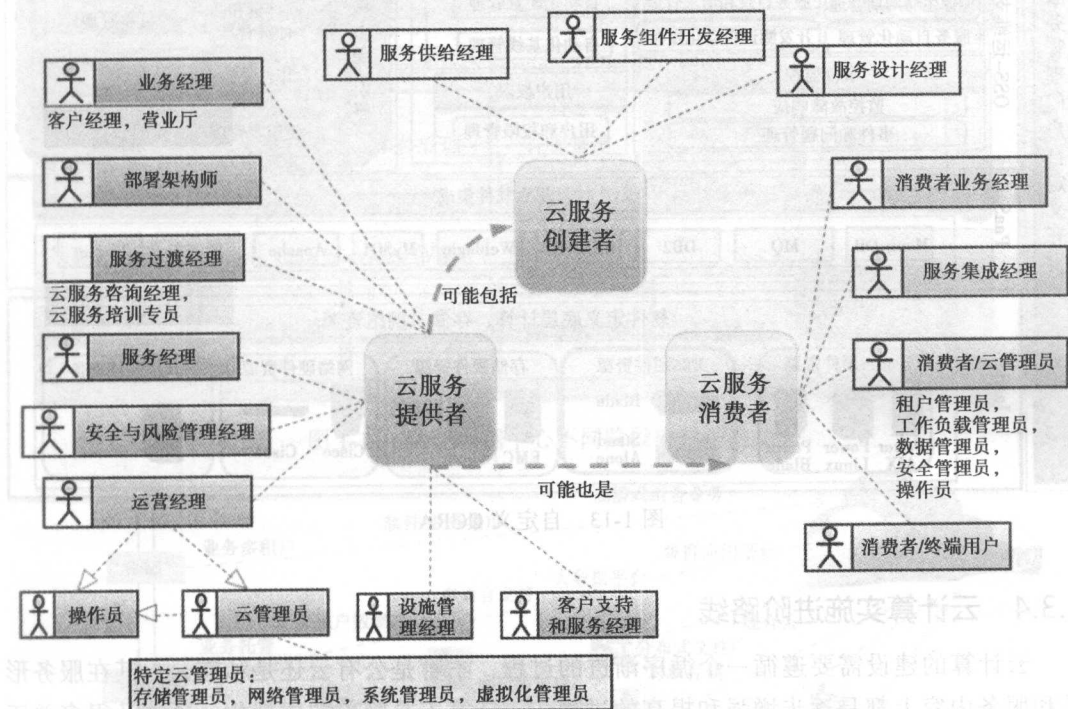


图 1-12 CCRA 云服务角色细分场景

对于大多数企业而言，云计算建设都不是一蹴而就的事情，整个项目的实施可能会经历多个阶段，这就需要企业明白每一个阶段应该完成哪些任务，这些任务完成之后，云化数据中心应该具备哪些功能。通常而言，云计算建设过程根据 CCRA 的定义，一般由下层到上层递进实施，当然如果人力资源和前期规划充分，也可以多个层次并行实施。就云计算而言，云化数据中心实施的第一步一定是底层基础架构设备的虚拟化，也可以称为资源池化，这一步就目前的虚拟化技术而言，已经相当成熟，功能组件也相当完善，主要需要考虑的还是虚拟化引擎的选择，以及如何整合企业目前已有的 IT 设备，而不是完全重新采

购。随着云化数据中心的成熟,各种自动化运维和监控以及IT资源的流程化管理和访问安全等功能应该被集成到云化数据中心,从而不断向真正成熟的云计算平台靠近。图1-14是云化数据中心在各个阶段应该具有的功能模块参考,其中有些功能模块是可以暂时不用考虑的(虚线模块),当然,如果企业急需某些高级功能,可以将这些功能模块往前移,并且如果认为某些功能模块完全没有必要,也可以将其剔除。

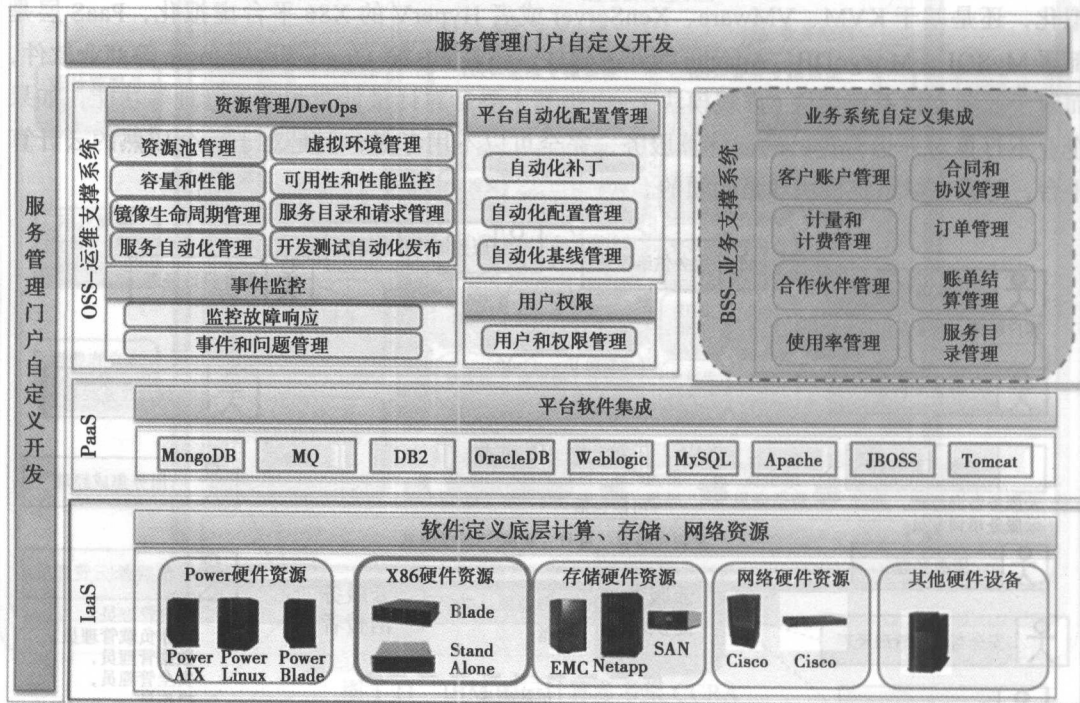


图 1-13 自定义 CCRA

1.3.4 云计算实施进阶路线

云计算的建设需要遵循一个循序渐进的过程,不管是公有云还是私有云,其在服务形态和服务内容上都是逐步增强和提高的过程。云计算不是简单的虚拟化,还涉及很多关于镜像、备份、存储、网络、访问控制、数据安全、高可用等相关领域的东西,而这其中的每个领域又都可以继续细分出多个专业领域,例如网络部分就有针对 OSI 七层模型中不同层次的高可用设计,以及 FWaaS、VPNaaS、LBaaS 等高层次服务的需求实现。在一个云环境中,一个企业甚至可以针对自己的专长,仅在某一领域深耕发展,例如安全和网络领域就已经很专业了。当然,对于一个成熟的多功能云化数据中心,多方向、多功能递进发展也是很有必要的,尤其是作为公有云运营商,不仅需要对内通过技术手段提升云服务的用户体验,还要不断提升防御各种日夜强化的外部恶意攻击。因此,走上了云,就意味着还有很长的路要走,而且需要循序渐进地往前走。图 1-15 是传统数据中心迈向云计算的过程

参考矩阵，从服务形态上来看，应该就是 IaaS 到 PaaS，然后到 SaaS 的过程；从服务内容上来看，从最初的虚拟化，随后进入运维管理及部署的自动化，最终实现智能化的过程。

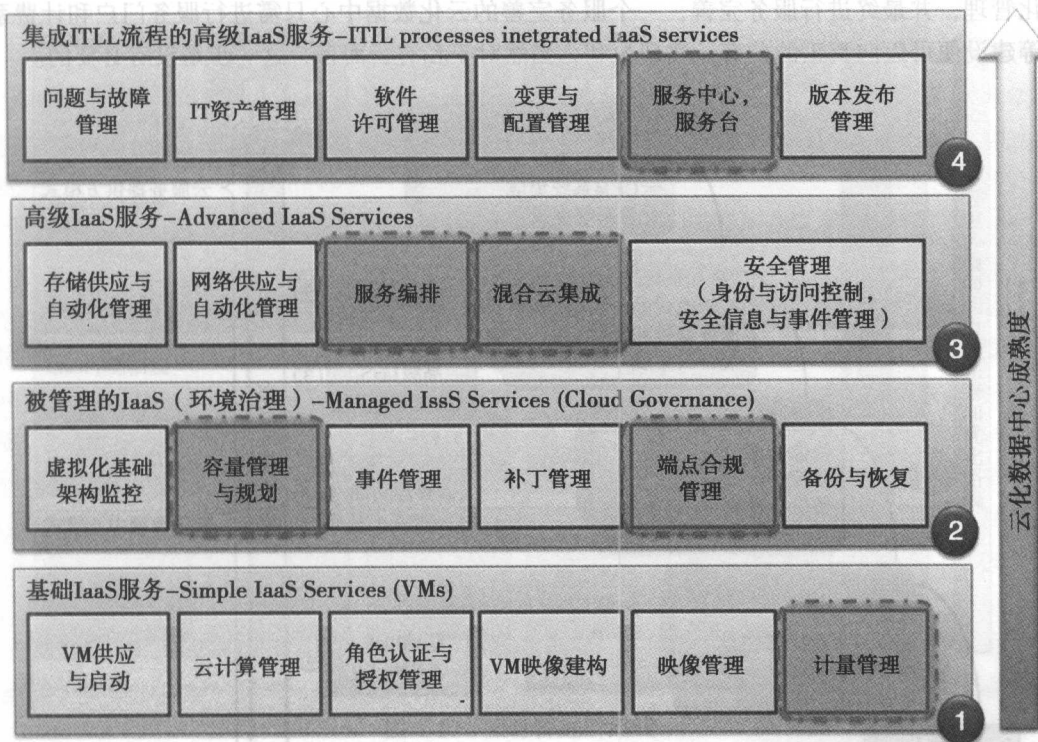


图 1-14 云化数据中心不同阶段应具备的功能

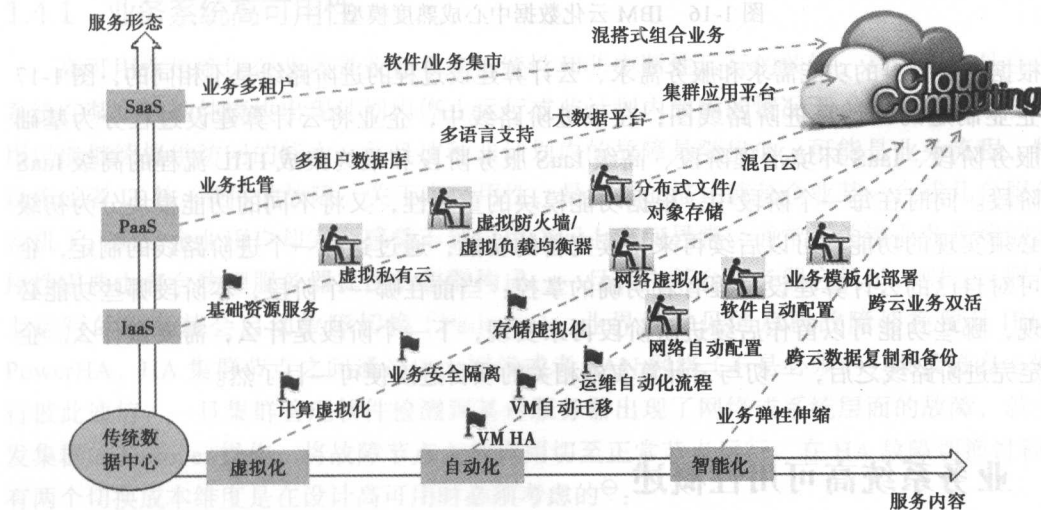


图 1-15 传统数据中心的云化过程参考矩阵

对于传统数据中心向云化数据中心的变迁，通常采用云化数据中心成熟度模型来衡量，

图 1-16 即为 IBM 提出的云化数据中心成熟度模型，这与图 1-14 描述的不同阶段云化中心应具备的功能模块是相互对应的。一个云化数据中心第一步便是虚拟化，然后进行自动部署和优化管理，并最终进行服务完善，一个服务完善的云化数据中心只需进行服务门户和计费系统等建设便可迈向真正的公有云实现盈利，当然对于私有云而言，这一步是没有必要的。

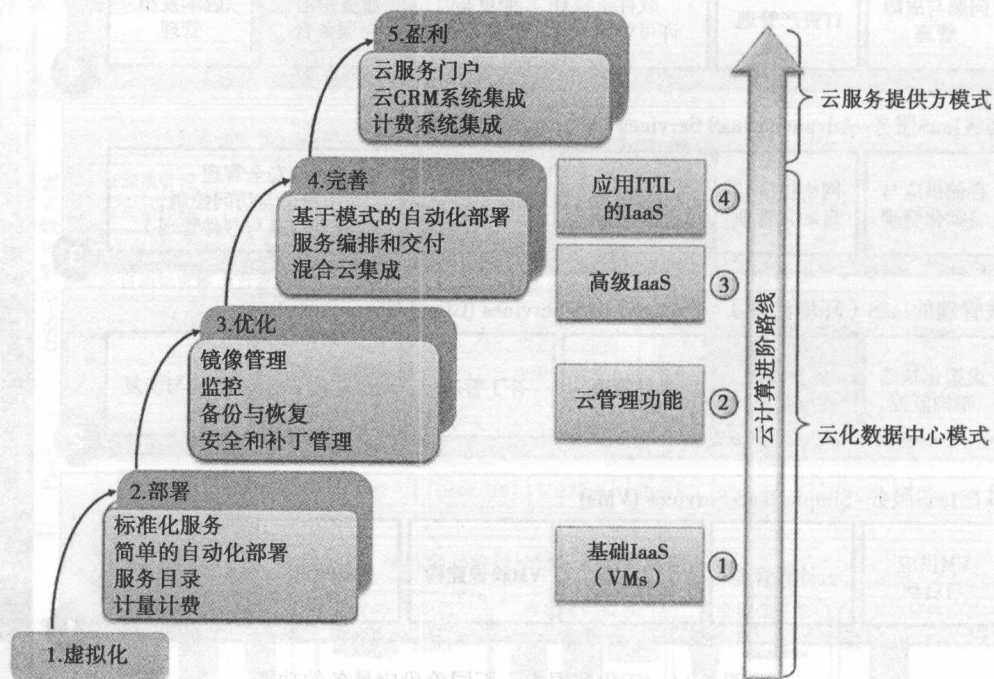


图 1-16 IBM 云化数据中心成熟度模型

根据企业自身的功能需求和服务需求，云计算建设过程的进阶路线是不相同的，图 1-17 是某企业制定的云计算进阶路线图，在该进阶路线中，企业将云计算建设过程分为基础 IaaS 服务阶段、IaaS 环境治理阶段、高级 IaaS 服务阶段和最终集成 ITIL 流程的高级 IaaS 服务阶段。同时在每一个阶段中，根据功能模块的重要性，又将不同的功能模块分为初级阶段必须实现的功能和可以后续再来持续完善的功能，通过这样一个进阶路线的制定，企业便可对自己的云计算建设过程有着明确的掌控：当前在哪一个阶段，本阶段哪些功能必须实现，哪些功能可以留作后续进阶阶段再来实现，下一个阶段是什么，需要做什么。企业制定完进阶路线之后，一切与云计算实施相关的项目进度便可一目了然。

1.4 业务系统高可用性概述

对于企业级用户而言，业务系统的高可用性（High Availability, HA）和容灾恢复（Disaster Recovery, DR），即通常所说的 HADR 是必须的，也是任何需要进入企业级应用

的 IT 架构无法回避的问题，不管是传统 IT 架构，还是新型的云计算架构，都要面临如何实现 HADR 的过程。

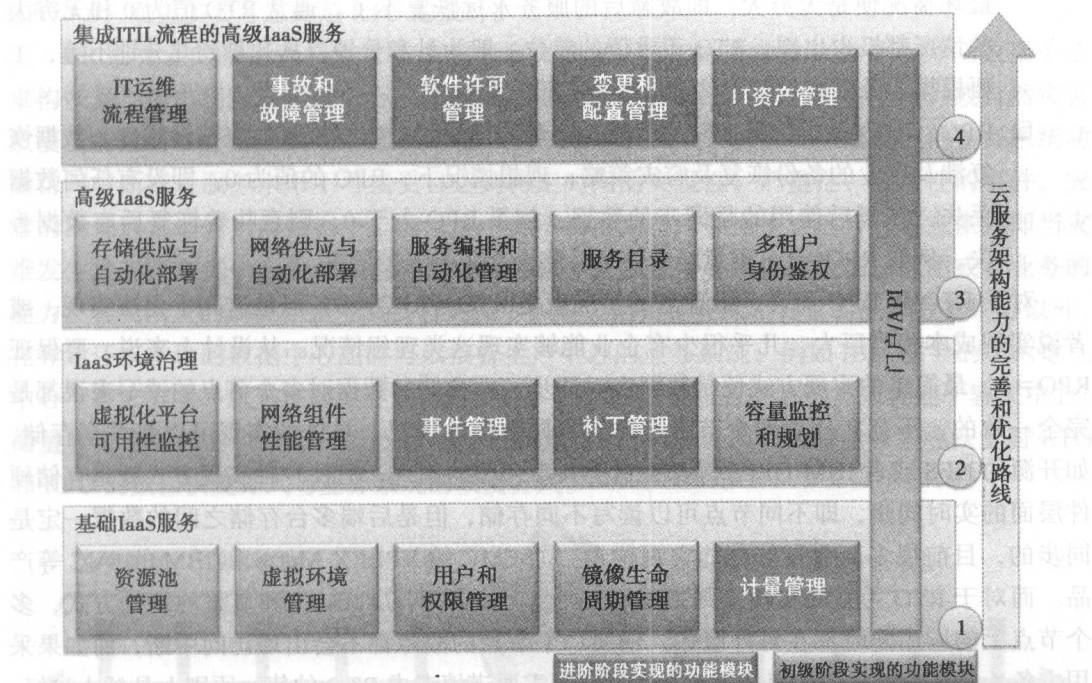


图 1-17 云计算进阶路线

1.4.1 业务系统高可用性

高可用性是确保企业重要业务系统持续性和非中断性运行的关键，高可用性是指本地系统在某个软硬件模块出现计划内停止运行或非计划内故障的情况下，基于本地系统的应用仍能继续提供访问的能力，并且这种非计划内的故障是随机的，可能是业务流程、物理设施或者 IT 软 / 硬件的故障。关于高可用性，最简单的描述就是企业某一台或几台服务器宕机了，但是企业用户却完全感觉不到应用访问上有任何异常。通常，关键业务系统的高可用性需要由多台物理服务器提供的集群构成，一旦其中某台服务器宕机了，则在该服务器上运行的服务就会启动故障切换（Failover）。业界最常见的双机故障切换便是 IBM 的 PowerHA，HA 集群节点之间通过以太网或者 SAN 网络（不是必须的）以及磁盘心跳进行彼此通信，一旦集群管理软件检测到某台服务器出现了网络或系统层面的故障，就会触发集群的 Failover 操作，将故障节点上的应用切至正常节点运行。在 HA 故障切换过程中，有两个切换成本维度是在设计高可用时必须考虑的^①：

① G. Venkata, G. Rajeev, E. Addison, et al. Implementing High Availability and Disaster Recovery in IBM Pure Application Systems V2 [J/OL]. IBM Red Book. 2015. <http://www.redbooks.ibm.com/>.

❑ RTO。RTO (Recovery Time Objective) 是指故障恢复的时间, 衡量的是故障恢复时间的快慢维度。RTO 的最佳值是 0, 即故障被立即恢复, 中间没有任何中断的时间; 最坏情况便是无穷大, 即故障后的服务永远恢复不了。通常 RTO 值为 0 和无穷大的情况都极少出现, RTO 正常值的单位一般为秒和分钟, 从几秒到几分钟不等, 主要根据业务系统的软硬件和集群架构设计来决定。

❑ RPO。RPO 是指数据恢复的程度, 衡量的是故障后数据恢复的完整性维度, 数据恢复涉及企业的备份恢复及容灾策略, 理想情况下, RPO 的值为 0, 即没有任何数据丢失, 恢复后使用的是同步的数据, 如果 RPO 大于 0, 则意味着恢复后有数据丢失, 例如 RPO=1, 则意味着恢复后将会丢失一天的数据。

对于 RTO 和 RPO 而言, 最理想的情况就是 $RTO=RPO=0$, 但是这几乎无法实现, 或者说实现成本相当巨大, 几乎很少有企业能够实现这类理想情况。从设计上来说, 要保证 $RPO=0$, 最简单的实现方式便是数据实时同步, 即存储的数据对多个节点的读写来说都是完全一致的, 不存在任意两个节点读取到不同数据的情况, 具体的实现可以是共享存储, 如开源的 NFS 或者 IBM GPFS 等都是这类共享存储的实现, 另外一种实现方式就是存储硬件层面的实时同步, 即不同节点可以读写不同存储, 但是后端多台存储之间的数据一定是同步的, 目前很多商业存储都能实现这点, 如 EMC 的 VPLEX Metro 和 IBM 的 SVC 等产品。而对于 RTO 为 0 的实现, 则是采用双活集群 (Active/Active) 和负载均衡的方式, 多个节点上的应用随时都在对外服务, 任何一个节点的故障都不会出现访问中断, 而如果采用主备 (Active/Passive) 模式的 HA 集群, 则需要谨慎考虑 RTO 的值, 原则上是越小越好。关于业务系统的高可用性, 一般通过全年的运行时间和宕机时间来计算, 也即我们经常在各种公有云运营商 SLA 上面看到的几个 9 的可用性, 高可用性的计算公式为: $[1 - (\text{宕机时间}) / (\text{宕机时间} + \text{运行时间})]$, 常见的主要有以下几个值^①:

- ❑ 1 个 9。即全年 90.0% 的可用性, 全年 365 天的宕机时间即为 36 天 12 小时。
- ❑ 2 个 9。即全年 99.0% 的可用性, 全年 365 天的宕机时间即为 87 小时 36 分钟。
- ❑ 3 个 9。即全年 99.9% 的可用性, 全年 365 天的宕机时间即为 8 小时 46 分钟。
- ❑ 4 个 9。即全年 99.99% 的可用性, 全年 365 天的宕机时间即为 52 分钟 33 秒。
- ❑ 5 个 9。即全年 99.999% 的可用性, 全年 365 天的宕机时间即为 5 分钟 35 秒。
- ❑ 11 个 9。这个几乎是几年才宕机一次了, 目前很少有云服务商能够做到这点, 更不用说传统的 IT 架构了, 当然 IBM 的 z 系列大型机例外。

1.4.2 业务系统容灾恢复

HA 与 DR 是有区别的, HA 更多的是强调本地系统的高可用, 即将某个应用系统运行在数据中心的多个服务器上, 当其中的任一服务器出现任意故障时, 应用程序和系统能迅

① Q. Dino, F. Steven, H. Mike, et al. High Availability and Disaster Recovery Planning: Next-Generation Solutions for Multiserver IBM Power Systems Environments [J/OL]. IBM Red Book. 2010. <http://www.redbooks.ibm.com/>.

速切换到其他服务器上运行从而保证业务系统的高可用性，其实现过程主要是本地系统的集群和数据的热备份，从设计上来讲，HA 往往通过共享存储来实现数据的同步，因此通常 $RPO=0$ ，而更多要考虑的是 RTO 的设计。

DR 则更多的是强调异地灾备中心的容灾恢复，即 DR 通常被认为是在另一数据中心重构恢复当前出现灾难数据中心的计划或过程。灾难（Disaster）是指由于人为或自然灾害致使当前数据中心内的 IT 系统受到严重破坏或者直接瘫痪，并最终导致相应的业务系统功能访问中断或者服务水平不可接受且达到特定时间的突发性、严重性、灾难性的事件，灾难的出现通常迫使当前数据中心的系统不得不切换到备用数据中心运行。容灾恢复即当灾难发生且生产数据中心受到严重程度破坏时在异地数据中心内恢复数据、应用或者业务的能力。容灾恢复的前提是企业具备容灾能力。容灾是指企业除了日常的生产数据中心以外，在异地还有备份的数据中心随时可以接管生产中心的业务系统，例如 IBM 倡导的“两地三中心”容灾方案，就是在同城建立备份中心，然后在地理位置更远的异地再建个容灾中心。衡量容灾系统的两个指标，仍然是 RTO 和 RPO。图 1-18 显示了影响业务系统恢复时 RTO 和 RPO 的数据恢复方式与业务系统的恢复方式。

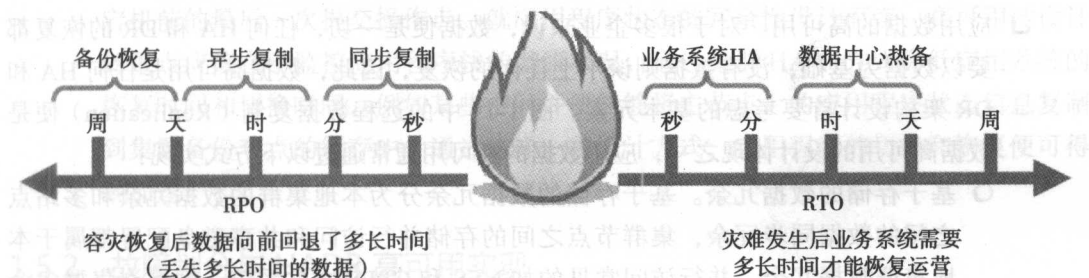


图 1-18 业务系统高可用的 RTO 与 RPO

从图 1-18 中可以看到，如果数据中心之间进行的是数据的同步复制，则容灾恢复过程中的 RPO 是秒级别的，即几乎不丢失数据，而如果是通过带库之类的备份恢复，则可能会丢失几天到几周不等的的数据。同时，如果业务系统建立有 HA，则恢复过程几乎是秒级别的，而如果是通过热备数据中心来恢复，则可能需要几个小时到几天不等。容灾恢复总体上可以分为数据级别、应用级别、业务级别：

- 数据级别。数据级别的容灾通常是建立异地容灾中心，通过数据的远程备份来实现，数据级别的容灾可以确保在灾难发生之后原有的数据不会丢失或者遭到破坏。但在发生灾难时应用是会中断的，因为数据级的容灾方式其实就是一个远程的数据备份中心，并不具有业务恢复的能力，此外，数据级容灾的恢复时间比较长，但是相比其他容灾级别来讲它的费用比较低，而且构建实施也相对简单，即 RTO 最长，总体拥有成本（TCO）最低。
- 应用级别。在数据级容灾的基础之上，在备份站点同样构建一套相同的应用系统，通过同步或异步复制技术在数据中心之间传递数据，这样可以保证关键应用在允许

的时间范围内恢复运行,尽可能减少灾难带来的损失,让用户基本感受不到灾难的发生,这样就使系统所提供的服务是完整、可靠和安全的。这一级别的 RTO 相对数据级别要优很多,同时 TCO 也相对较小。

- 业务级别。几乎就是生产数据中心的模板复制,全部业务系统都做了应用级别的灾备,同时除了必要的 IT 相关人员和技术,还要求具备全部的基础设施。在严重灾难发生后,原有的办公场所都会受到破坏,在业务级别的容灾环境下,除了数据和应用的恢复,业务系统的正常开展也要被恢复。当然,这一级别的容灾恢复 RTO 是最低的,同时 TCO 是最昂贵的。

1.5 传统 IT 架构高可用设计

1.5.1 传统数据中心 HADR 设计原则

任何 HADR 设计架构都需要遵循一定的高可用设计原则,对于一个成熟完善的 HADR 高可用解决方案,其架构设计必须包含以下基本模块^①:

- 应用数据的高可用。对于很多企业来说,数据便是一切,任何 HA 和 DR 的恢复都要以数据为基础,没有数据则谈不上任何的恢复,因此,数据高可用是任何 HA 和 DR 架构设计首要考虑的基本元素。图 1-19 中的远程数据复制 (Replication) 便是数据高可用的设计体现之一,应用数据的高可用通常通过以下方式实现:
 - 基于存储的数据冗余。基于存储的数据冗余分为本地集群的数据冗余和多站点之间的数据同步冗余,集群节点之间的存储并行访问和共享磁盘配置都属于本地集群数据冗余,并行访问常见的如 NFS 和 GPFS 文件系统,都允许集群多个节点同时读写存储。共享磁盘配置常见的如 IBM 的 PowerHA 共享盘,这是一种 Active-Passive 的访问模式,只有在 Active 的节点故障的情况下,磁盘锁才会被释放同时 Passive 节点才能读写共享盘。跨站点数据复制分为基于存储的数据复制和基于服务器端的数据复制。服务器端的远程数据复制主要通过镜像 (Mirror) 技术来实现;存储端的数据复制又分为同步复制 (Synchronous Replication) 和异步复制 (Asynchronous Replication),异步复制意味着主节点应用程序可以继续执行而不用等待存储端的数据复制完成,同步复制则需要等待远程存储写完成的应答信号,因此同步复制的应用程序响应性能不如异步复制,但是同步复制在故障恢复时不存在数据丢失的情况,而异步复制无法做到这点。
 - 基于日志的数据复制。基于日志的数据复制功能主要应用在数据库的容灾恢复上,如 IBM 的 DB2 数据库便有 HADR 功能,其主要思想是数据库可以通过记录的操作日志来进行数据恢复。因此,备份了操作日志,便间接地备份了保存在

① Q. Dino, F. Steven, H. Mike, et al. High Availability and Disaster Recovery Planning: Next-Generation Solutions for Multiserver IBM Power Systems Environments[J/OL]. IBM Red Book.2010. <http://www.redbooks.ibm.com/>.

数据库中的数据。

- 应用基础架构的高可用。基础架构设施为应用的正常运行提供了全部的软硬件资源，基础架构的高可用体现在两个方面，其一是提供在集群节点中重新启动应用所需的全部软硬件资源，其二便是通过监控和验证确保集群的完整性。应用基础架构的高可用最常见的便是双机主备或者双机互备集群环境，两个节点从物理硬件到操作系统软件和参数配置上都保持一致，主节点故障后，应用切换到备节点运行，由于备节点拥有与主节点几乎完全一样的环境，因此应用的重启不存在任何问题。
- 应用运行状态的高可用。在传统企业级业务系统中，很多应用是有运行状态的，如果要确保应用正常恢复，则需要保存应用的运行状态，并将恢复后的应用从故障前的最新状态点重新运行。应用程序的运行状态通常保存在内存中，而应用程序的恢复点依赖于你的应用程序设计和故障类型等很多环境因素，以 DB2 数据库为例，DB2 每进行一次提交操作（Commit），则内存中的数据便会写入磁盘，数据库认为此次数据操作正常完成，而如果用户还没有提交便出现了宕机事件，则恢复后的应用（DB2 数据库）不会从宕机时刻的状态开始重新启动，而数据库的恢复点应该是宕机前的最后一次提交操作点。就应用程序状态的冗余性设计而言，高可用性设计的重点应该在于监控应用程序栈的健康状况，通过集群 HA 的方式降低应用系统的恢复时间和切换时间，例如某些中间件便能够将主节点上的应用程序状态信息复制到集群备份节点的缓存中，通过这样一种设计方式，应用程序的切换和恢复便可得到加速。

1.5.2 故障划分与 HADR 高可用实现

在传统 IT 领域，高可用设计通常利用设备的冗余热备和软件的集群方式来实现，如 IBM 的 z 系列大型机便是利用每个关键部件的冗余设计来保证硬件层面上的高可用，当然这种大型机的设计思路也被应用到了很多小型机里面，包括 IBM 的 POWER 系列小型机和 HP 的 Superdome 等企业级服务器和存储设备。传统 IT 领域的高可用性设计主要分为硬件层面和软件层面的设计，而根据 IBM 的调查，硬件故障引起的宕机只占了小部分比例，接近 50% 的宕机事件是由软件问题和人为误操作引起的。而高可用性设计所要解决的问题便是引起业务系统宕机的各种故障，然后根据这些故障的特点进行针对性的高可用性设计。表 1-2 总结了可能引起宕机事件的故障组，这些故障组是高可用设计时需要考虑的首要因素。

表 1-2 高可用设计故障归类

故障组归类	HADR 需要考虑的故障
故障组 1	中央处理器相关模块（CPU、内存）、I/O 板卡、磁盘驱动等故障
故障组 2	网络及存储相关的适配卡故障，线路故障等

(续)

故障组归类	HADR 需要考虑的故障
故障组 3	关键操作系统资源：卷、文件系统、IP 等
故障组 4	应用，中间件，管理员误操作等
故障组 5	数据中心故障（电力、空调、灾难等）

在业务系统出现故障的时候，HA 有助于降低故障引起的业务中断时间，同时促使关键资源在高可用集群服务器之间进行可靠的故障切换，而在多站点容灾恢复的场景中（DR），高可用集群解决方案除了能够增强业务的高可用性之外，还能够管理和保障站点之间的数据复制，因此集合 HA 与 DR 的 HADR 高可用容灾解决方案将会是企业业务系统真正实现高可用和持续性运行的理想解决方案，并且 HADR 几乎可以覆盖表 1-2 中的全部故障组。一个典型的 HADR 设计案例如图 1-19 所示^①。

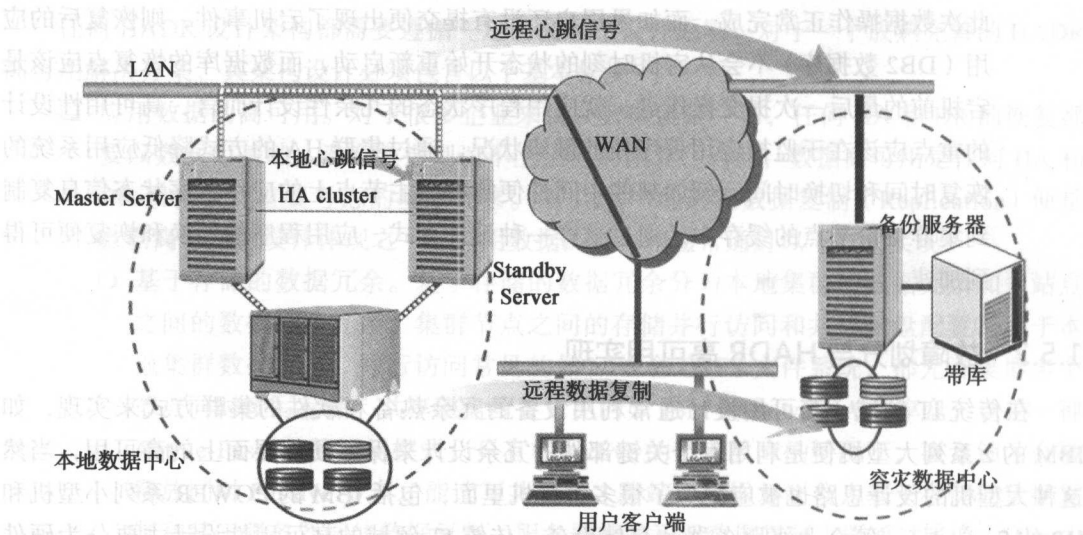


图 1-19 HADR 设计案例

图 1-19 中，本地数据中心与远程数据中心之间通过远程数据复制的方式实现数据容灾，本地数据中心内部构建 HA 集群，同时除了本地 HA 集群之间需要建立心跳检测机制之外，本地数据中心与远程数据中心之间也需要实现相应的心跳检测机制。如果本地数据中心内部的 Master Server 出现故障，则本地 HA 集群将触发资源切换（Failover），如图 1-20 所示。

图 1-20 中发生 HA 故障触发资源切换时，并不会触发容灾恢复，此时远程数据复制仍然进行，数据中心之间的站点心跳也保持正常进行，此时如果本地数据中心出现灾难性事故，导致整个数据中心 IT 系统无法使用，则会立即触发 DR 容灾恢复过程，本地数据中心

^① <http://www.cnblogs.com/sammyliu/p/4741967.html>

业务系统将全部迁移至容灾数据中心，或者说容灾数据中心将利用平时同步复制的数据进行业务系统的恢复，如图 1-21 所示。

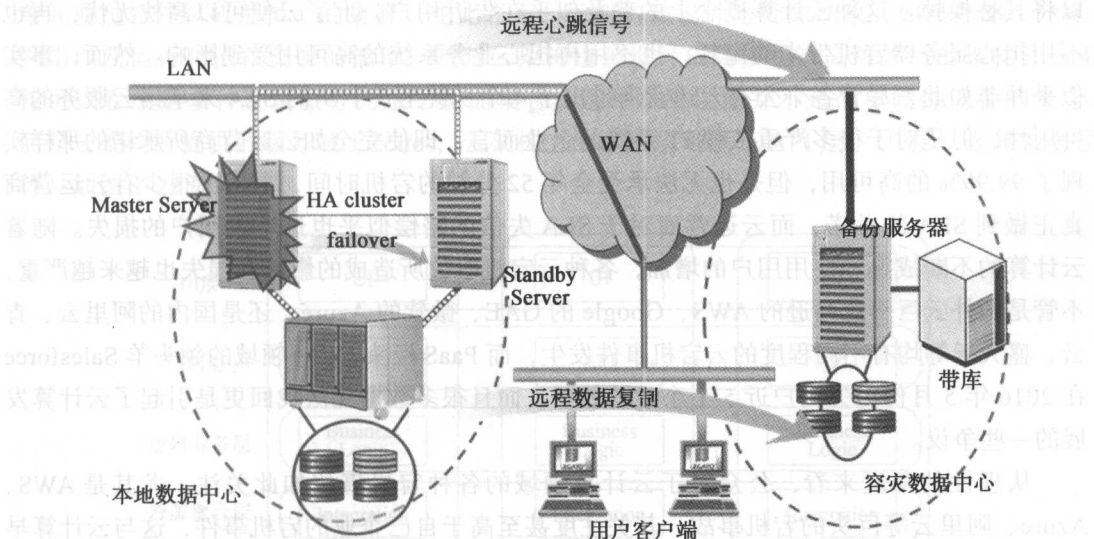


图 1-20 HADR 架构设计中的 HA 切换

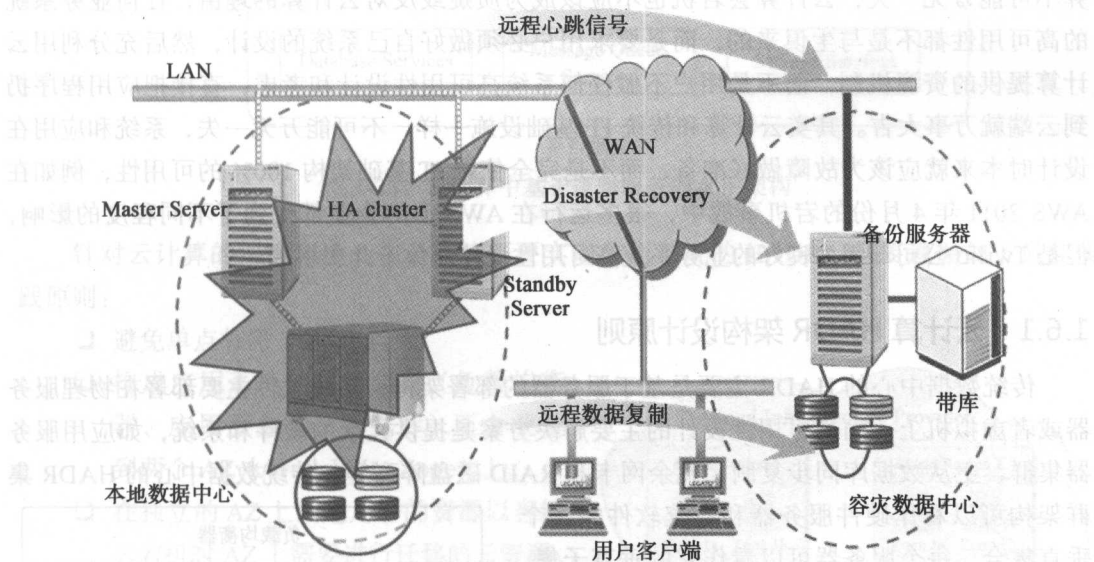


图 1-21 HADR 架构设计中的 DR 切换

1.6 云环境下的高可用设计

在云计算布道的阶段，关于如何对待云计算环境下服务器与传统数据中心服务器的最

形象的比喻也许就是“宠物”与“绵羊”了。在传统数据中心里，用户总是小心翼翼地维护如宠物般珍贵的服务器，而在云环境下，服务器就像放养的绵羊，出了任何问题随时可以将其替换掉。这种云计算概念上的普及似乎在告诉用户，上了云便可以高枕无忧，再也不用担心服务器宕机带来的噩耗，也不用再担心业务系统的高可用受到影响。然而，事实似乎并非如此，尽管各个公有云运营商推出了4个9甚至5个9的SLA来承诺云服务的高可用性，但是对于很多严重依赖IT系统的企业而言，即使完全如云运营商所承诺的那样实现了99.99%的高可用，但是也无法承受全年52分钟的宕机时间，更何况很少有云运营商真正做到SLA的承诺，而云运营商对于SLA失信的赔偿似乎也远不及用户的损失。随着云计算的不断成熟和使用用户的增加，各种云宕机事故所造成的影响和损失也越来越严重，不管是国外云巨头亚马逊的AWS、Google的GAE、微软的Azure，还是国内的阿里云、青云、盛大云等均有不同程度的云宕机事件发生，而PaaS巨头CRM领域的领头羊Salesforce在2016年5月份丢失用户近5个小时的数据，而且很多数据无法找回更是引起了云计算发展的一些争议。

从理性的角度来看，公众对于云计算领域的各种宕机事故如此关注，尤其是AWS、Azure、阿里云等巨头的宕机事故，其关注度甚至高于自己企业的宕机事件，这与云计算早期的宣传致使很多用户对云计算抱有100%高可用的不切实际的期望有关。事实上，云计算不可能万无一失，云计算会宕机也不应该成为质疑或反对云计算的理由，任何业务系统的高可用性都不是与生俱来的，而是要求用户必须做好自己系统的设计，然后充分利用云计算提供的资源机制，而不是用户不做任何系统高可用性设计和考虑，直接把应用程序扔到云端就万事大吉。其实云计算和传统IT基础设施一样，不可能万无一失，系统和应用在设计时本来就应该为故障做好准备，而不是完全依赖IT基础架构100%的可用性，例如在AWS 2011年4月份的宕机事故中，很多运行在AWS上的企业都受到了不同程度的影响，但是Twilio公司却因为良好的业务系统高可用性设计而免于此次事故。

1.6.1 云计算 HADR 架构设计原则

传统数据中心的HADR主要是基于服务器的部署架构，系统软件主要部署在物理服务器或者虚拟机上，而高可用性设计的主要解决方案是提供冗余的硬件和系统，如应用服务器集群、主从数据库同步复制、冗余网卡和RAID磁盘阵列等。传统数据中心的HADR集群架构可以看作硬件服务器和系统软件的一个垂直集合，每个服务器可以看作它的垂直子集，集群中任何一台服务器故障，其他服务器仍然可以响应应用请求，如图1-22所示。

与传统数据中心基于服务器的架构不同，对于企业而言，云计算环境下最大的技术转换便是云的出现只是为了提供服务，即云架构的



图 1-22 传统数据中心基于服务器的应用架构

核心应该是如何呈现和利用云服务。传统数据中心需要提供运行时的设备，如应用和数据库服务器，并且是以独立且不可再分的服务器单元提供，然后这些独立不可再分的服务器单元通过软件组成逻辑上的集群。相比之下，云架构被设计为资源抽象，如对存储、计算、网络以及运行时依赖资源的抽象，同时云架构也被设计成为多租户高可扩展的架构。云架构可以看成是多种逻辑服务的水平层面集合，这种水平层面并不是简单硬件设备的集合，而是云架构所提供的服务域（Zone），云架构上的水平层可以是 API 层、UI 层、逻辑业务层等，如图 1-23 所示。

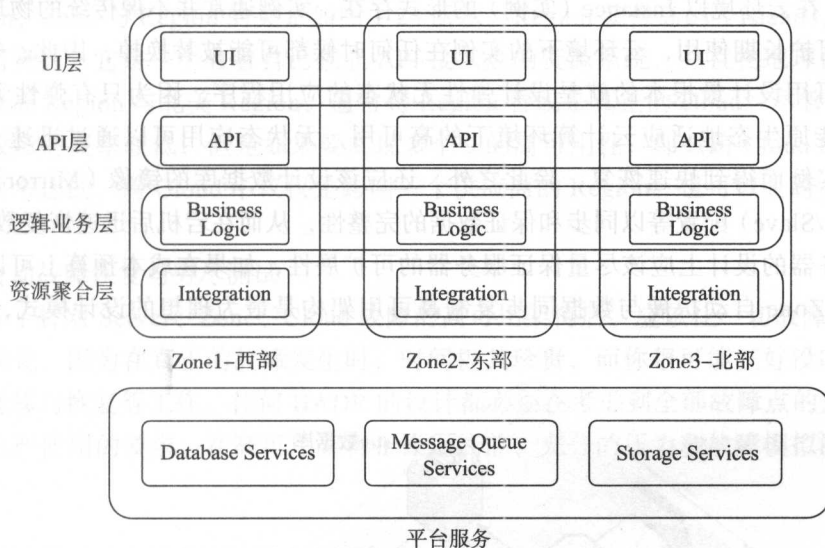
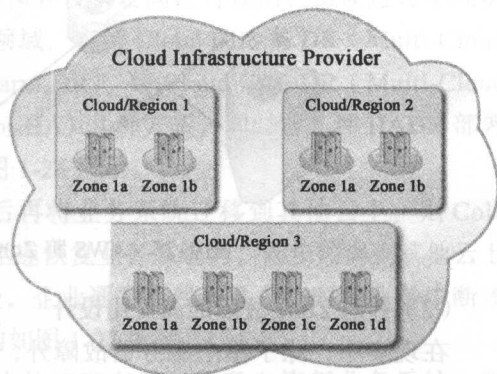


图 1-23 云计算中基于逻辑服务的水平架构

针对云计算的架构环境，在云环境下要实现应用系统的 HADR，需要遵循如下最佳实践原则：

- ❑ 避免单点故障（SPOF）。
- ❑ 构成应用系统的每个组件（负载均衡器、应用服务器、数据库）至少放置到两个 AZ（Availability Zone）上。
- ❑ 在独立的 AZ 上预建足够的资源以容纳云宕机时 AZ 上需要进行迁移的云资源。
- ❑ 跨多个 AZ 进行数据同步复制或者跨区域（Region）进行数据备份以应对云宕机时进行应用程序的 Failover，AZ 与 Region 的区别如图 1-24 所示。



- ❑ 设置监控、告警及故障识别和自动解

图 1-24 可用域（AZ）和区域（Region）的关系

决或自动 Failover 的运行机制。

❑ 构建弹性无状态的应用程序以便灾难发生时在安全的 AZ 内进行实例 reboot 或者 relaunch 以恢复应用正常运行。

上述最佳云计算 HADR 实践原则总结起来可以归为 4 个步骤,即为 Server 故障设计高可用、为 Zone 故障设计高可用、为 Cloud/Region 故障设计高可用、自动故障恢复与穷尽测试:

(1) 针对 Server 故障的高可用设计

Server 在云环境以 Instance (实例) 的形式存在,实例通常并不像传统的物理服务器一样被小心呵护长期使用,云环境下的实例在任何时候都可能被替换掉,因此,针对 Server 级别的高可用设计最根本的就是设计弹性无状态的应用程序,因为只有弹性无状态的应用程序才能原生态地适应云计算环境下的高可用,无状态应用可以通过迅速 relaunch 或者 reboot 实例而得到快速恢复,除此之外,还应该设计数据库的镜像 (Mirror) 备份、主从 (Master/Slave) 配置等以同步和保证数据的完整性,从而在宕机后迅速恢复数据可用性,在应用服务器的设计上应该尽量保证服务器的可扩展性,如果在成本预算上可以接受,则 AWS 的跨 Zone 自动扩展与数据同步复制高可用架构是最为理想的设计模式,如图 1-25 所示。

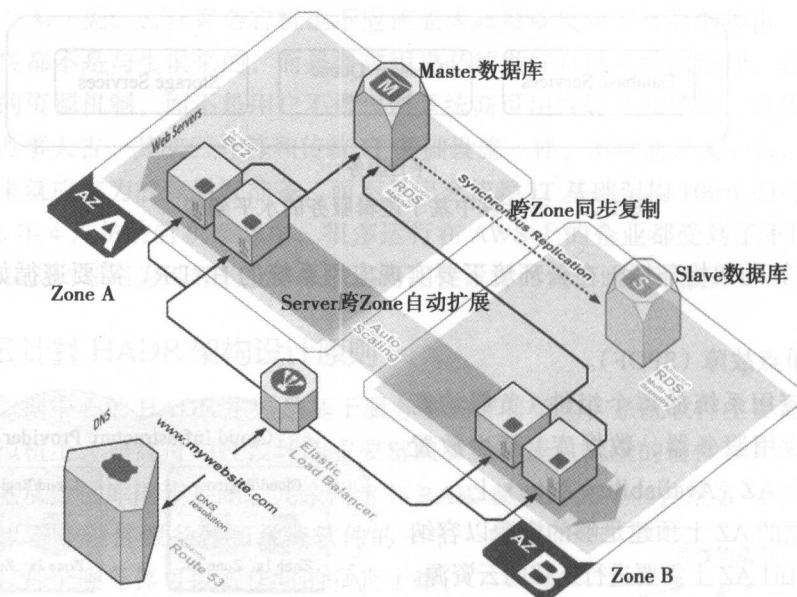


图 1-25 AWS 跨 Zone 自动扩展与数据同步架构^①

(2) 针对 Zone 故障的高可用设计

在现实中,除了单个服务器故障外,还可能会出现数据中心的电源故障、网络故障、

^① <http://blog.csdn.net/awschina/article/details/27505541>.

闪电雷击以及人为施工破坏等数据中心的故障，因此还需为你的应用考虑 Zone 故障。Zone 是指一个使用独立电源，并且与其他 Zone 彼此隔离的数据中心，Zone 内的 Server 共享高带宽、低延时、可通过私有网络访问的局域网。应对 Zone 故障的方案就是将每一个应用层的 Server 都扩展到至少两个 Zones 上，数据也需要进行至少跨两个 Zone 的复制，图 1-25 中的 AWS 高可用架构便是如此。

(3) 针对 Cloud/Region 的高可用设计

从 AWS 的观点来看，Servers 集群不是云，独立 Zone 也不是云，云是由多个 Zone 组成的 Region，Zone 与 Region 的关系如图 1-24 所示。Region 就像岛屿 (island)，彼此之间不共享任何资源，它是一个有着自己独立 API 接入点的资源系统，并且在地理位置上相距很远，如亚洲 Region 与北美 Region，通常将 Region 看成是真正的 cloud。尽管一个 Cloud 发生出现故障的概率较低，但是如果要实现多个 9 的高可用性，那么跨 Cloud 的高可用设计也是需要考虑的，跨 Cloud 不仅仅是跨同一个供应商的 Region，也可以是跨多个供应商的 Region。

(4) 自动故障恢复与穷尽测试

在设计了针对 Server、Zone、Cloud 故障的高可用架构后，应该让一切故障的 Failover 都变得自动化，因为在真正的故障发生时，时间极其珍贵，而你很可能正好没时间应对复杂的数据迁移与恢复等工作。任何 HADR 的设计都必须在考虑到全部故障点的充分测试下才能符合生产使用的要求，在高可用架构正式上线前，充分的压力和故障模拟测试是必经的过程。

1.6.2 云计算 HADR 架构设计实现

对于 HADR 而言，永远都是方案越完善成本越昂贵，HADR 与成本预算总是正相关，因此，企业在真正考虑自身业务系统的高可用设计时，不可能完全按照最完美的 HADR 方案来实现，很多中小企业和初创企业更是如此。因此，企业在设计高可用云计算方案时，应该在自身可以承受的高可用级别和能够接受的预算成本之间进行权衡，从而进行不同层次的 HADR 设计考虑。在云计算环境的 HADR 领域，有跨 Cloud 的冷备 DR (Multi-Cloud Cold DR)、跨 Cloud 半热备 DR (Multi-Cloud Warm DR)、跨 Cloud 热备 DR (Multi-Cloud Hot DR) 以及跨 Cloud 热备 HA (Multi-Cloud Hot HA) 几种方式，同时这几种 HADR 部署架构的 RTO/RPO 与预算成本成正相关关系，如图 1-26 所示。

如果企业仅希望降低成本，在云宕机出现后再将业务系统迁移到其他云上，则 Cold DR 方案是最佳选择，但是 Cold DR 方式是不能迅速恢复业务系统的，同步数据到其他云上也很缓慢，要将数据库启动到运行状态也很缓慢，企业通常要经历几个小时的业务中断才能重新恢复，以 AWS 部署为例，Cold DR 的架构如图 1-27 所示。

如果企业希望以最小的额外成本和尽可能快的速度在另外的云上恢复业务系统，则 Warm DR 方案是最佳选择。在这种方案下，备份云中心运行 Slave 数据库的服务器，同时

数据被实时复制到备份云中心，一旦主中心出现云宕机事故，便可在备份云中心迅速启动实例恢复应用系统，Warm DR 的部署架构如图 1-28 所示。

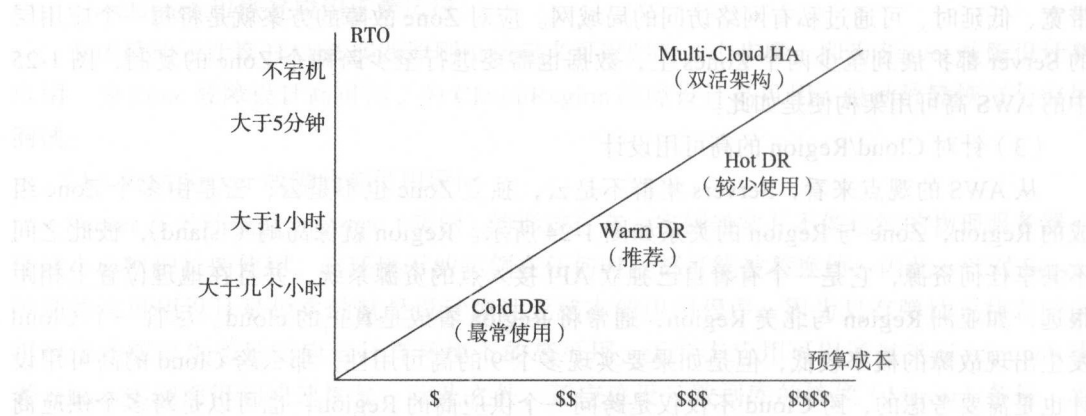


图 1-26 云架构 HADR 与成本关系[⊖]

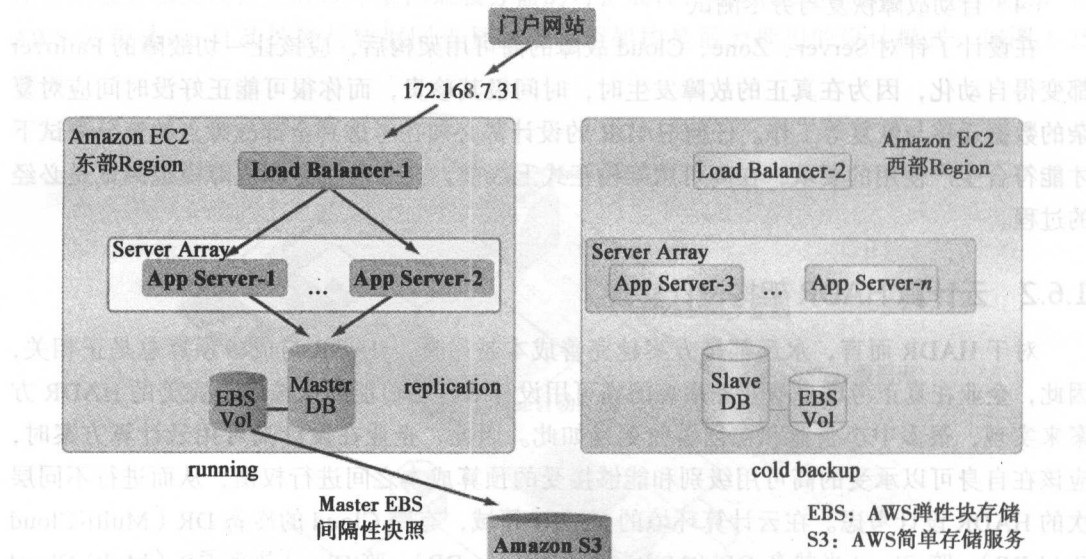


图 1-27 Cold DR 架构

Warm DR 方案在灾备云中心并不同时运行实例，而只进行数据层面的备份与同步，相比之下，Hot DR 方案不仅在灾备云中心进行数据层面的保护，还同时运行同等规模的应用实例，但是灾备云中心的实例并不负载客户的访问请求，一旦主中心出现云宕机事故，则灾备中心的云实例迅速接管用户访问请求。与 Warm DR 方案相比，Hot DR 方案显然要支付更多的费用，并且在云环境与传统 IT 数据中心不同，启动应用实例的速度本身已经非常

[⊖] <http://docplayer.net/1520360-High-availability-in-the-cloud-architectural-best-practices.html>.

快，因此 Hot DR 的恢复时间不会比 Warm DR 高很多，但是费用却要昂贵得多，因此并不推荐 Hot DR 的方案，Hot DR 的架构如图 1-29 所示。

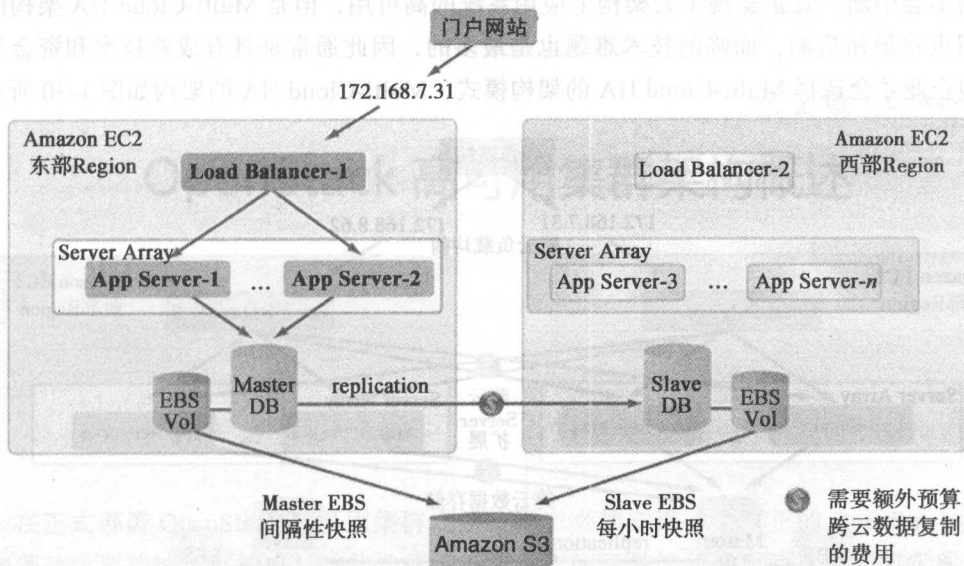


图 1-28 Warm DR 架构图

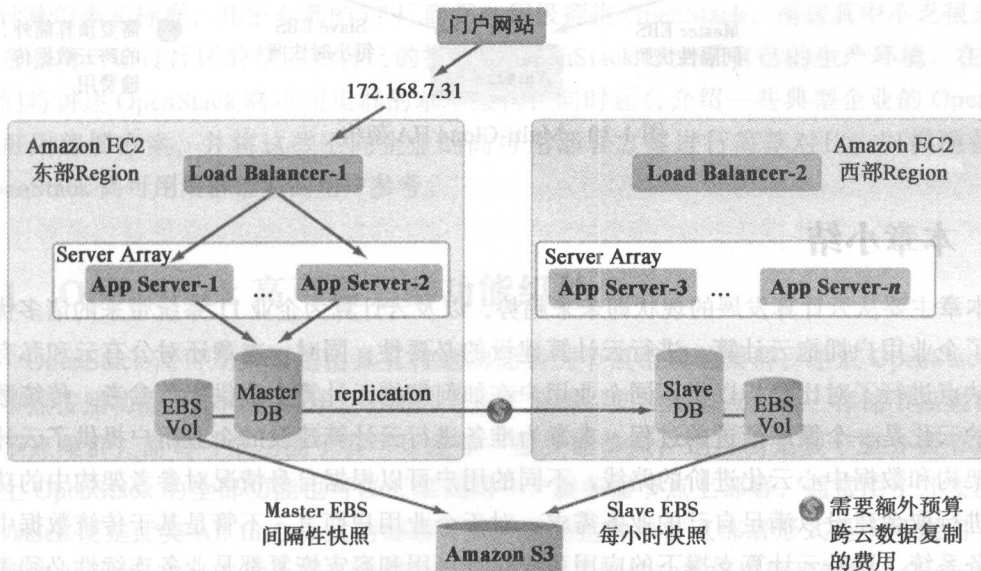


图 1-29 Hot DR 架构

跨云 DR 方案虽然可以保证企业应用的恢复，但是始终存在中断时间，对于大型企业，尤其是金融领域的企业而言，业务中断是不能允许的，因此跨云 HA（Multi-Cloud HA）将是最佳选择，Multi-Cloud HA 类似于传统数据中心的双活方案，主备云中心均同时运行同

一套应用系统，并通过负载均衡器实现跨地理位置的负载均衡，打通云之间的网络后实现应用服务器的跨云扩展部署和数据存储，这样即使主备云中心出现云宕机事故，企业应用系统均不会中断，真正实现了云架构上应用系统的高可用，但是 Multi-Cloud HA 架构的成本费用也将最昂贵的，面临的技术难题也是最多的，因此通常是具有成熟技术和资金实力的大型企业才会选择 Multi-Cloud HA 的架构模式，Multi-Cloud HA 的架构如图 1-30 所示。

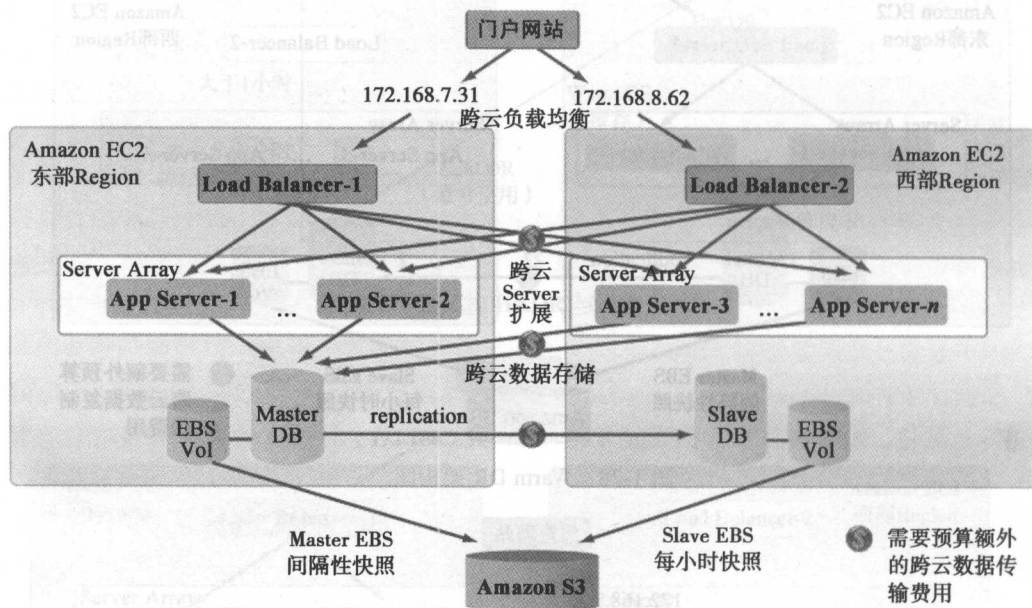


图 1-30 Multi-Cloud HA 架构

1.7 本章小结

本章主要从云计算发展的现状即未来趋势，以及云计算为企业 IT 系统带来的诸多优势阐述了企业用户拥抱云计算、进行云计算建设的必要性，同时，本章还对公有云和私有云的优缺点进行了对比分析以供不同企业用户在如何拥抱云计算上提供决策参考。传统数据中心的云化是一个循序渐进的过程，本章为准备进行云计算建设的企业用户提供了云计算参考架构和数据中心云化进阶的路线，不同的用户可以根据自身情况对参考架构中的功能模块进行取舍建设以满足自己的业务需求。对于企业用户而言，不管是基于传统数据中心的业务系统，还是云计算支撑下的应用系统，高可用和容灾恢复都是业务连续性必须考虑的问题，本章不仅对传统 IT 架构下的高可用和容灾恢复进行了介绍，还重点介绍了云计算时代，用户业务系统的高可用架构和实现参考。总体而言，本章主要是云计算和业务系统高可用的架构及实现的介绍，是后续 OpenStack 高可用集群建设的基础，在后面的章节中，我们将会通过 OpenStack 开源云来对本章所讲解的内容进行具体的实现介绍。

OpenStack 高可用集群架构概述

在正式部署 OpenStack 高可用集群之前，用户必须知道一个真正的 OpenStack 集群应该由哪些主要功能组件构成，这些功能组件彼此之间又是如何传递消息和如何实现高可用的。OpenStack 到目前为止已经经历了近七年的发展，在这七年之中，OpenStack 成为了开源云计算的事实标准，几乎全部的 IT 厂商都在积极拥抱 OpenStack，而这其中不乏很多大中型企业用户通过社区的帮助和自己的摸索将 OpenStack 推上了自己的生产环境。在本章，我们将讲述 OpenStack 高可用集群的基本架构，同时还会介绍一些典型企业的 OpenStack 高可用部署方案，并将这些不同企业的高可用部署方案进行简单对比，以供准备进行 OpenStack 高可用集群部署的用户参考。

2.1 OpenStack 高可用集群功能组件

OpenStack 高可用集群是由具有特定功能集节点组成的集群，组成 OpenStack 集群的节点按照功能的不同被划分为不同的节点子集群，如控制节点集群、存储节点集群和计算节点集群，而每一个功能子群可以是单一服务器节点，也可以是多个服务器节点，理论上 OpenStack 的全部功能也可以汇聚到同一个服务器节点上部署，通常用于开发测试的 devstack 便是此类 All In One 式的部署形式，但是这种单节点部署形式是不可能用于生产环境的。在 OpenStack 的生产环境部署中，通常将不同的功能模块称为组件（Component），例如计算组件、存储组件和网络组件，而这些上层抽象功能集组件又可以细分为更具体的功能组件，同时对于生产环境而言，OpenStack 中的每一个组件都应该具备高可用性。如果从更为抽象的层次来描述 OpenStack 集群，则可以将集群划分为组成各个功能集的服务器节点和连接服务器节点以传输不同数据和指令的网络。

在 OpenStack 的高可用集群生产环境部署中, OpenStack 集群是由执行多种不同功能角色的物理(或者虚拟机)服务器节点组成的。在 OpenStack 集群中, 某些特定的集群功能角色可能由特定的服务器来执行, 同时, 其他的功能角色可能会由分布在其上的多台服务器共同执行, 例如 Nova-api 主要由控制节点执行, 而 Nova-compute 却可以同时运行在控制节点和多台计算节点上。

2.1.1 集群控制节点

OpenStack 控制节点主要运行 API 服务, 规划调度 (Scheduler) 服务, 以及其他一些辅助但是 OpenStack 正常运行所必需的服务, 如数据库、对象缓存系统、消息队列系统等, OpenStack 集群的控制节点管理了集群组件之间的全部日常活动, 同时响应客户终端和开发人员发出的指令请求并进行资源分析调度, 因此可以认为控制节点是整个 OpenStack 集群的核心大脑, 也是 OpenStack 集群需要重点对其进行高可用设计的部分, 因为控制节点的故障将会导致整个 OpenStack 集群无法使用。从服务的角度来看, OpenStack 控制节点上的每个服务既可以跨越部署在全部控制节点, 也可以独立部署在部分控制节点, 但是在高可用生产环境中, 全部 OpenStack API 服务、Scheduler 服务、Neutron-Server 服务等均运行在全部控制节点上, 同时作为存储 OpenStack 集群配置参数和数据的 MySQL 或 MariaDB 数据库也通过集群的形式部署在全部控制节点上, 作为集群 RPC 通信的高级消息队列 RabbitmqMQ 也需要部署在全部控制节点上, 除此之外, 控制节点还需要以高可用的方式运行 Memcached 和 Redis 缓存系统。在高可用集群部署中, 控制节点可以看成是整个集群的控制面板, 随时监控 OpenStack 集群全部资源的运行情况并进行资源的启停、恢复和迁移等操作, 为此, 控制节点需要运行某种集群资源管理器。在传统的 Linux 高可用部署和目前的 OpenStack 高可用集群部署中, 最为成熟的集群资源管理器 (Cluster Resource Manager, CRM) 便是 Pacemaker 和 Corosync 的组合, 因此在 OpenStack 集群的高可用部署中, 全部控制节点上还需运行 Pacemaker 和 Corosync 进程, 以进行节点心跳通信和资源管理控制。任何运行在控制节点集群中的服务都可以分为 Active/Active 模式、Active/Passive 模式和 Active/Hot-Standby 高可用模式, 具体每个服务应该以哪种高可用模式运行在控制节点上, 主要取决于服务的状态模式, 即服务是有状态 (Statefull) 还是无状态服务 (Stateless)。

通常, 在需要部署 MySQL Galera 集群的生产环境中, 都要求控制节点集群至少由三个服务器节点构成, 不过对于 OpenStack 的 API 服务、RabbitMQ 消息队列服务、弹性虚拟 IP 服务和负载均衡器的高可用而言, 两个服务器节点其实已经足够, 这里要求控制节点由三个服务器节点构成, 主要是 Galera 集群和 Pacemaker/Corosync 集群的正常运行需要 Quorum 机制的支持。在 OpenStack 控制节点集群中, 有状态服务和无状态服务的高可用部署是有较大差异的, 例如 OpenStack API 这类无状态服务全部被部署成 Active/Active 高可用模式, 而数据库服务通常部署为 Active/Passive 模式, 图 2-1 为部分服务在三控制节点模

式下的高可用部署情况。

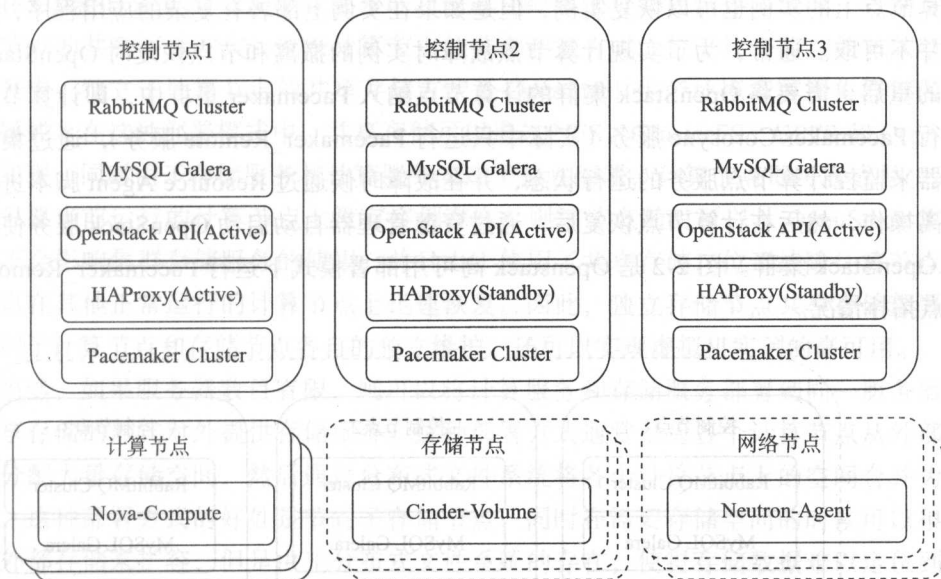


图 2-1 三控制节点服务高可用部署模式

2.1.2 集群计算节点

计算节点是真正运行各种 Hypervisor 虚拟化管理软件的节点，也是 OpenStack 集群中真正提供计算能力的节点。计算节点为虚拟机实例（Instance）运行提供了所需要的 CPU、内存、网络等资源，在 OpenStack 的默认部署模式下，计算节点上运行的是基于内核的虚拟化引擎 KVM，KVM 是一种集成到 Linux 系统内核的开源 Hypervisor，通过 KVM 虚拟化后的计算节点能够组成庞大的计算资源池，从而对外提供计算资源服务，而计算节点上的实例通过 KVM 访问底层的硬件资源。在 OpenStack 的集群部署中，计算节点天然具有强大的可扩展性，如果新的计算节点需要加入 OpenStack 集群以扩充集群的计算能力，则只需将新加入的节点向 Nova 的 Scheduler 服务注册即可（安装启动 Nova-compute 服务后便可自动注册），然后 Nova 的 Scheduler 服务就会使用一定的资源过滤算法（如 RoundRobin 算法）来在集群中全部已注册的计算节点中选择某个计算节点以响应客户端创建实例的请求，然后被选中的计算节点便会根据控制节点传送来的实例请求参数进行实例创建。就 OpenStack 高可用集群的部署而言，计算节点的访问请求并不需要负载均衡，Nova 的 Scheduler 服务会自动屏蔽故障不可用的计算节点，并且还会根据一定算法来选择能够满足请求资源数目的计算节点（或者剩余资源最多的计算节点）进行实例创建，针对计算节点的高可用，真正要考虑的是如何应对或者迁移故障计算节点上的虚拟机（VM），并在故障计算节点重新启动后自动恢复与 OpenStack 相关的服务（如 Nova-compute 等），从目前的各种 OpenStack 虚拟机高可用方案来看，Nova 项目提供的实例撤离（Evacuate）和在线迁移（Live-Migration）是使

用较多的方案,此外,利用 heat 项目的模板功能在其他正常计算节点上快速重建 (Rebuild) 故障计算节点上的实例也可以恢复实例,但是如果在实例上部署有复杂的应用程序,则此种方法并不可取。通常,为了实现计算节点故障时实例的撤离和节点恢复时 OpenStack 相关服务的重启,需要将 OpenStack 集群的计算节点纳入 Pacemaker 集群中,即计算节点也需要运行 Pacemaker/Corosync 服务 (实际中只运行 Pacemaker_Remote 服务),通过集群资源管理器来监控计算节点服务的运行状态,并在故障时候通过 Resource Agent 脚本进行实例的撤离操作,然后在计算节点恢复后,通过资源管理器自动启动 OpenStack 服务使其重新加入 OpenStack 集群。图 2-2 是 Openstack 高可用部署模式下运行 Pacemaker_Remote 的计算节点拓扑情况。

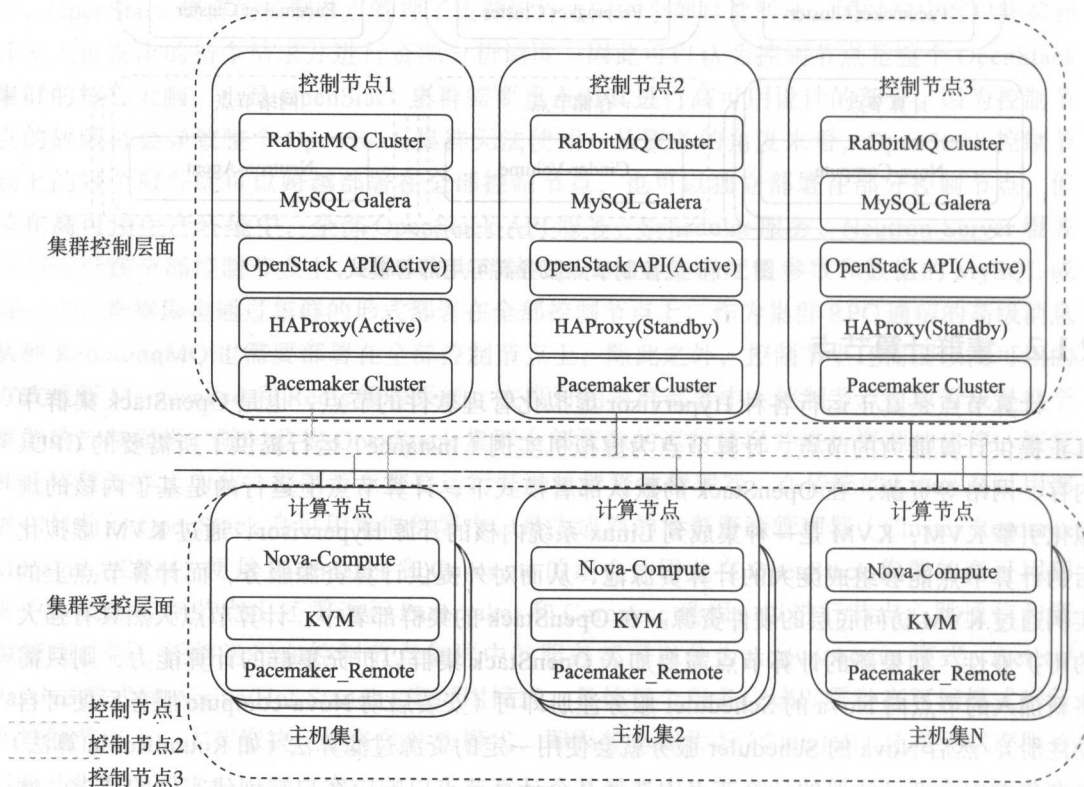


图 2-2 OpenStack 计算节点高可用部署

2.1.3 集群存储节点

在 OpenStack 集群中,有很多服务项目组件需要使用后端存储来存储数据,如 Nova 项目需要存储实例,Glance 项目需要存储镜像,而在 OpenStack 集群中提供存储服务的主要是块存储组件 Cinder 和对象存储组件 Swift,以及可以同时提供文件系统、块存储、对象存储服务的 Ceph 集群。OpenStack 集群中存储节点的部署比较灵活,用户可以根据自身情况

将存储节点与计算节点分开独立部署，也可以将存储服务 and 计算服务同时部署到一个节点上。在 OpenStack 集群中，存储服务可以通过三种形式对外提供存储服务，分别是独立于计算节点的共享存储方式、位于计算节点的共享存储方式、位于计算节点的非共享存储方式。其中，独立于计算节点的共享存储方式是 OpenStack 生产环境高可用集群部署推荐的部署模式。在这种部署模式中，共享存储可以通过 NFS、GlusterFS、GPFS 和 Ceph 集群的形式实现，同时由于存储服务与计算服务实现了独立部署，计算节点即可看成是“无状态”的，只要当前运行的计算节点上没有实例存在，则该节点随时可以从 OpenStack 集群中移除而不会影响集群存储服务的使用，并且由于使用了共享存储，位于故障计算节点上的实例可以在其他正常运行的计算节点上迅速恢复。因此，独立存储节点共享存储部署模式不仅有利于计算节点和存储节点各自的独立维护，还可以实现虚拟机实例的高可用。

当然，如果服务器数目有限，则可以将计算服务和存储服务部署到同一服务器上，并以共享存储的形式对外提供存储服务。这种部署方式通常是各个计算节点从外部存储设备上分配大量存储空间，然后通过分布式文件系统将各个计算节点上的空间合并为单一挂载点。这种部署方式的好处是节省了存储节点，同时在需要存储空间的时候可以很方便地通过外部存储来扩容，但是由于分布式文件系统的存在，使得存储数据难以定位到具体的节点，并且可能会出现存储服务和计算服务同时争抢计算节点资源的情况而导致节点性能下降。另外一种部署模式是存储服务与计算服务合并但是使用共享存储，即每个存储节点独立在计算节点上，计算节点仅能使用位于其上的存储服务。这种部署模式的优点在于节省存储节点的同时，独立计算节点上的高 I/O 并不会影响其他计算节点上的实例性能，但是缺陷也很多，比如计算节点故障后，实例无法在其他计算节点上恢复，也很难实现实例在计算节点之间的迁移，同时需要更多存储空间时，无法实现在线的存储扩容。

在大规模高可用 OpenStack 集群部署中，由于 Ceph 拥有自己的集群高可用和数据恢复机制，并且易于扩展、维护和满足各种存储对象的需求，因此 Ceph 集群通常被用于替代 Cinder 和 Swift 的后端存储功能，简单来说，Ceph 集群由 Ceph 监控服务、Ceph OSD 服务以及 Ceph 客户端构成，在 Ceph 与 OpenStack 的整合部署中，Ceph 的 Monitor 服务被部署到控制节点上，而 Ceph 的 OSD 进程可以独立于计算节点，也可以合并到计算节点上部署，Ceph 的客户端通常是 Nova 的 Compute 服务和 Glance 以及 Cinder 服务，图 2-3 为 Ceph 与 OpenStack 整合后的高可用部署模式。

2.1.4 集群网络节点

网络节点为租户提供了基于用户自定义的虚拟网络服务，这些服务包括虚拟网桥、虚拟交换机、DHCP 服务器、L3 虚拟路由等。OpenStack 项目中早期的网络功能由 Nova 的 Network 服务提供，之后独立为 Quantum 项目，并最终演变为 Neutron 项目。在 OpenStack 官方推荐的集群部署中，Neutron 的服务组件分布在控制节点、计算节点和网络节点上：控制节点上运行 Neutron-server 服务以响应租户的网络服务请求；计算节点上运行 Openv-

switch 等代理服务以负责计算节点上虚拟机对外部网络或者租户子网之间的通信；网络节点上运行二层网桥和三层路由服务，负责整个 OpenStack 集群租户网络的二层转发和三层路由等功能。在实际生产环境的大规模高可用集群部署中，网络节点的服务高可用性和性能瓶颈是部署难点也是网络部署所面临的挑战，由于 Neutron 的 L3 服务是“有状态”服务，因此不能简单地通过负载均衡的形式实现高可用，同时如果将整个 OpenStack 集群的网络全汇聚到网络节点，则随着规模的扩大，网络节点必然成为整个集群的性能瓶颈。针对 L3 的高可用，最常见的方法是基于虚拟冗余协议（VRRP）的 L3 HA 和基于分布式虚拟路由（DVR）的高可用网络部署方案，L3 HA 可以解决三层路由的高可用问题，但是不能解决网络瓶颈问题，而 DVR 虽然可以解决网络瓶颈问题和 L3 的目的地址切换（DNAT）高可用问题，但是遗留了源地址切换（SNAT）的单点故障问题。

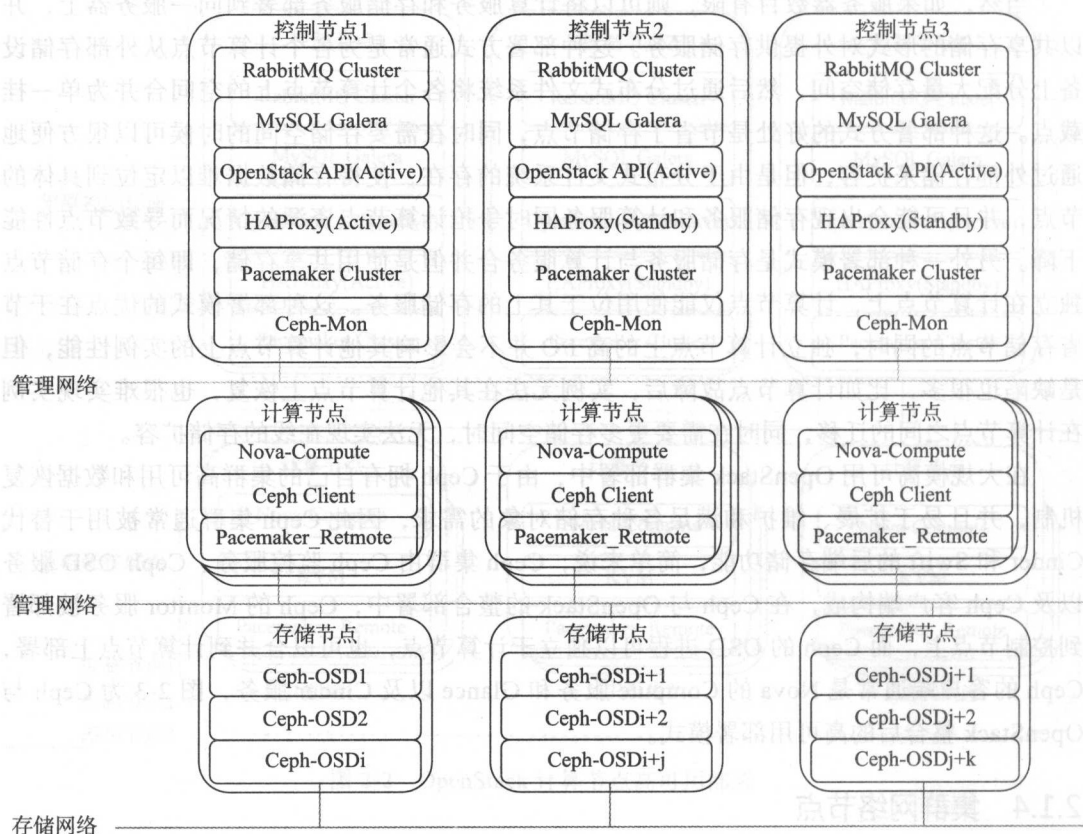


图 2-3 OpenStack 高可用集群部署中的 Ceph 集群

在 OpenStack 集群的高可用部署中，网络节点的功能通常被部署到控制节点上，通过控制节点的 Pacemaker 集群资源管理器利用相应的资源代理（Resource Agent）实现 Neutron 服务的高可用性，即将网络节点与控制节点合并，利用控制节点的冗余和 Pacemaker 集群来提供 Neutron 服务的高可用，在这种部署模式下，网络节点的角色仍然存在，只是由控

制节点来负责实现。另一种在 OpenStack 大规模高可用集群部署中可选的网络部署模式是 DVR 模式，在 DVR 下网络节点唯一的功能就是进行 SNAT，而剩余的网络节点功能全被部署到各个计算节点上，每个计算节点上的网络组件负责本节点上虚拟机的网络通信，从而实现网络功能的分布式和高可用。在 Mitaka 版本中，实现了 DVR 与 SNAT HA 的集成部署，因此在 Mitaka 及以后版本中，最理想的部署模式就是将网络节点的 SNAT 功能部署到控制节点集群上实现高可用，网络节点剩余的功能通过 DVR 分布到各个计算节点上，图 2-4 为 L3 HA 高可用集群网络部署模式，图 2-5 为 DVR 集成 SNAT HA 的高可用集群网络部署模式。

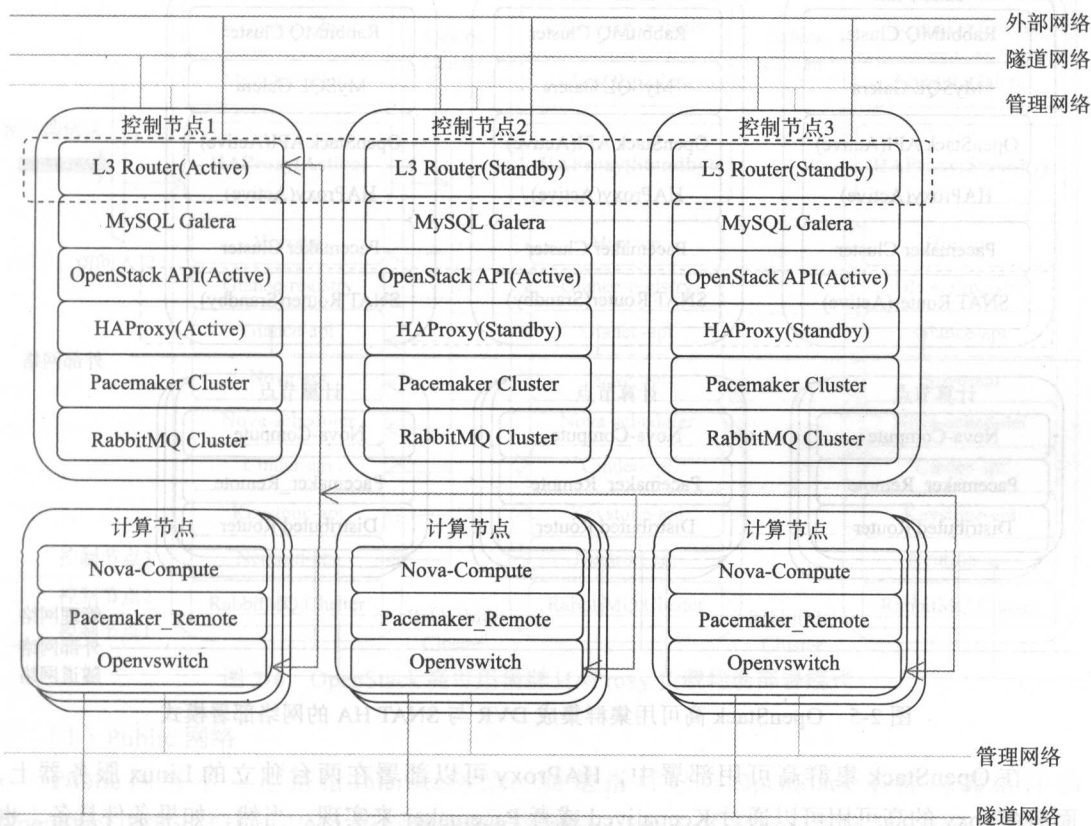


图 2-4 OpenStack 高可用集群 L3 HA 网络部署模式

2.1.5 集群负载均衡器

负载均衡器 (LoadBalancer) 是极为常见的应用程序高可用组件，在 OpenStack 集群高可用部署中，负载均衡器通常是指 HAProxy 软件，其作用是将对 OpenStack 相关服务的 HTTP/TCP 访问请求负载均衡到两个或者多个后端服务控制节点上，每个控制节点上均部署相同的组件，各个控制节点上相同服务组件彼此之间组成 Active/Active 或者 Active/

Passive 高可用服务模式，HAProxy 结合后端控制节点的健康状态和均衡算法来决定服务器请求应该转发到哪个节点，因此，某个后端控制节点的故障并不会影响到 OpenStack 集群的对外服务。在 OpenStack 集群服务的访问过程中，HAProxy 扮演了集群服务入口的角色，任何对后端 OpenStack 服务的访问请求都需要经过 HAProxy 的代理转发，也即 OpenStack 客户端是无法直接访问位于负载均衡器后端控制节点上服务的，客户端的请求必须经过负载均衡器转发才能访问 OpenStack 集群服务。

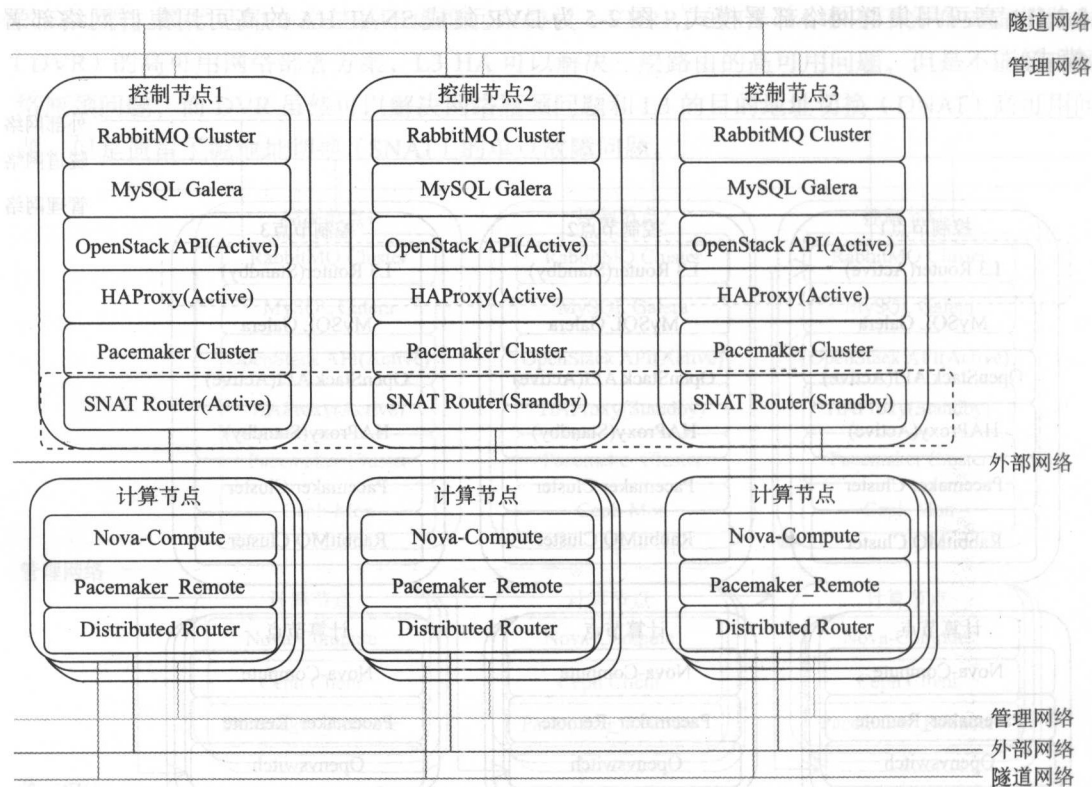


图 2-5 OpenStack 高可用集群集成 DVR 与 SNAT HA 的网络部署模式

在 OpenStack 集群高可用部署中，HAProxy 可以部署在两台独立的 Linux 服务器上，而 HAProxy 的高可用可以通过 Keepalived 或者 Pacemaker 来实现。当然，如果条件具备，也可以直接使用硬件负载均衡器，如 F5 交换机来实现集群访问的负载均衡。在通常的多控制节点 OpenStack 集群部署中，为了便于集群资源的统一管理，可以将 HAProxy 直接部署到多个控制节点上，并通过控制节点上的 Pacemaker 集群实现 HAProxy 的高可用性，图 2-6 是三控制节点 OpenStack 高可用集群中的 HAProxy 部署模式。

2.1.6 集群网络拓扑

OpenStack 集群部署中会使用到不同角色的网络，不同角色网络负责各自的数据传输，

从而实现网络功能独立运行以提高整个集群的性能。在某些特定情况的部署中,平时相对空闲时间较多的功能网络可以共用同一个物理网卡,而承载较大数据传输的网络使用专有网卡。专有网卡根据业务数据量大小可以采用单个高带宽网卡,也可以采用多个高带宽网卡绑定组合后的逻辑网卡。如果是针对高性能计算的 OpenStack 集群,则数据传输网络还可以采用 Infiniband 等高速交换网络。本节主要介绍 OpenStack 集群部署中将会使用到的各种网络拓扑。

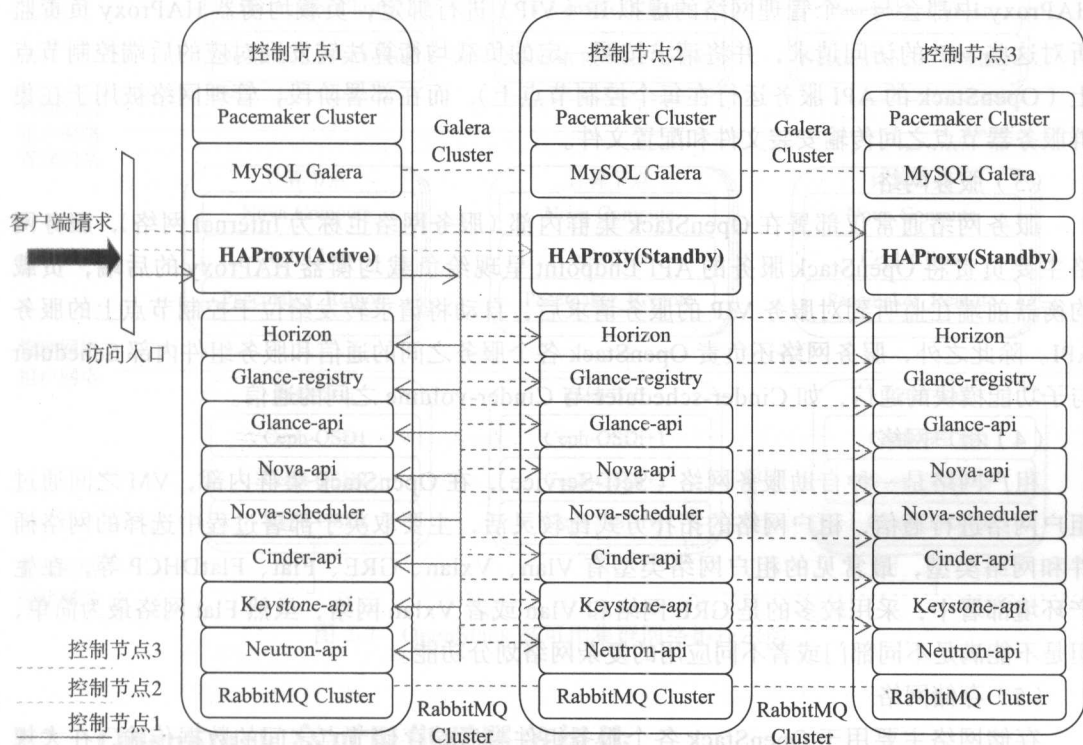


图 2-6 OpenStack 高可用集群 HAProxy 负载均衡部署模式

(1) Public 网络

Public 网络不一定是 Internet 网络,而是指可以与 OpenStack 集群内部组件和 OpenStack 内部虚拟机 (VM) 进行通信的外部网络 (Public 网络有时也称为 External 网络),它可以是 Internet 网络,也可以是公司内部使用的办公网络或者业务网络。Public 网络为 OpenStack 集群中的 VM 提供了可路由的外部网络 IP 地址空间,在 OpenStack 官方推荐的控制节点、计算节点、网络节点和存储节点独立部署的环境中,网络节点作为 OpenStack 集群的 VM 与外界通信的网络转发服务器,将会使用 Public 网络的 IP 地址来进行 Source NAT 和 Destination NAT 操作,从而实现 VM 到外部网络 (如 Internet) 的通信和租户对 OpenStack 集群内部 VM 的访问。在 OpenStack 集群的生产环境高可用部署中,通常使用 L3 HA (L3 服务部署到多个控制节点上) 或者 DVR 部署模式,而在 DVR 模式下,每个计

算节点都会接入 Public 网络从而实现 VM 与外部网络的通信。

(2) 管理网络

管理网络在 OpenStack 集群中主要用于初期的系统软件安装部署和后期的运行维护,如果是私有云环境,则公司内部管理人员可以通过管理网络对 OpenStack 集群进行访问。通常 OpenStack 高可用集群的部署中,管理网络主要的两个功能角色是负载均衡和集群节点之间的安装配置文件传输。在 OpenStack 的高可用集群中,每个 OpenStack 服务在 HAProxy 中都会与一个管理网络的虚拟 IP (VIP) 进行绑定,负载均衡器 HAProxy 负责监听对这些 VIP 的访问请求,并将请求按照一定的负载均衡算法转发到对应的后端控制节点上 (OpenStack 的 API 服务运行在每个控制节点上),而在部署阶段,管理网络被用于在集群服务器节点之间传输安装文件和配置文件。

(3) 服务网络

服务网络通常仅部署在 OpenStack 集群内部 (服务网络也称为 Internal 网络),服务网络主要负责将 OpenStack 服务的 API Endpoint 呈现给负载均衡器 HAProxy 的后端,负载均衡器前端在监听到对服务 VIP 的服务请求后,自动将请求转发给位于控制节点上的服务 API。除此之外,服务网络还负责 OpenStack 各个服务之间的通信和服务组件内部 Scheduler 与子功能模块的通信,如 Cinder-scheduler 与 Cinder-volume 之间的通信。

(4) 租户网络

租户网络是一种自助服务网络 (Self-Service),在 OpenStack 集群内部,VM 之间通过租户网络进行通信,租户网络的拓扑方式比较灵活,主要取决于部署过程中选择的网络插件和网络类型,最常见的租户网络类型有 Vlan、Vxlan、GRE、Flat、FlatDHCP 等,在生产环境部署中,采用较多的是 GRE 网络和 Vlan 或者 Vxlan 网络,虽然 Flat 网络最为简单,但是不能满足不同部门或者不同应用的复杂网络划分功能。

(5) 存储网络

存储网络主要用于 OpenStack 各个服务组件与后端存储节点之间的数据传输,在大规模 OpenStack 集群部署中,尤其是 I/O 密集型集群中,存储数据网络极有可能成为应用集群性能的瓶颈,因此,在集群部署初期应该谨慎规划存储数据网络。在以 Ceph 作为后端存储的 OpenStack 集群部署中,必须为 Ceph 集群指定专有的数据传输网络,而集群监控服务可以共用管理网络,如果条件具备,Ceph 集群内部的数据网络采用 Infiniband 交换网络将会极大提高数据传输性能。

在实际的部署过程中,并不要求各个网络之间保持严格的独立性,如果没有太多的 IP 地址资源,则管理网络、服务网络、租户网络可以合并为同一个网络来进行部署,但是承载相应网络负载的网卡需要有较高的带宽,高带宽的逻辑网卡可以通过多张物理网卡进行链路聚合等方式实现,图 2-7 为 OpenStack 集群高可用部署环境下的网络拓扑示例。

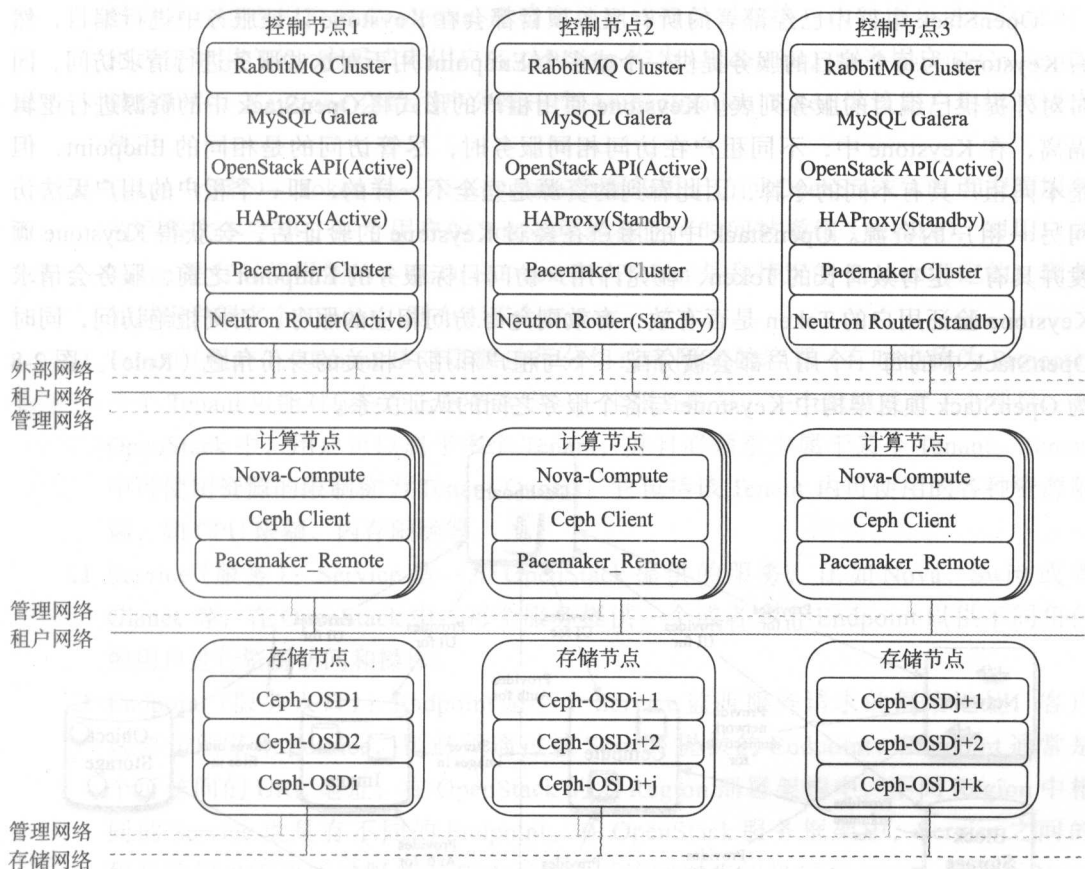


图 2-7 OpenStack 高可用集群网络拓扑示例

2.2 OpenStack 高可用集群服务组件

2.2.1 认证服务 Keystone

OpenStack 集群中所有用户与服务的集中认证授权机制通过 Keystone（OpenStack Identity 服务）项目来实现，Keystone 在 OpenStack 服务框架中负责用户身份验证、服务目录编目和身份令牌的管理功能，它实现了 OpenStack 的 Identity API。Keystone 类似一个服务总线，或者说是整个 OpenStack 服务框架的注册表，其他服务通过 Keystone 来注册其服务并获得该服务的 Endpoint，即该服务的访问入口（服务访问的 URL），此外，任何服务之间的相互调用，也要经过 Keystone 的身份验证才能获得目标服务的 Endpoint 以访问目标服务。Keystone 支持多种形式的认证授权，包括常用的用户名和密码验证系统、基于 Token 的令牌验证系统和使用 Public/Private 密钥对的验证系统，除此之外，Keystone 还可以集成普遍使用的目录服务（Directory Service），如轻量级目录访问协议（LDAP）。

OpenStack 集群中已经部署的所有服务项目都会在 Keystone 认证服务中进行编目，然后 Keystone 为每个编目的服务提供一个或多个 Endpoint 用于对这些服务进行请求访问，同时对外提供已编目的服务列表。Keystone 使用租户的形式将 OpenStack 中的资源进行逻辑隔离，在 Keystone 中，不同租户在访问相同服务时，尽管访问的是相同的 Endpoint，但是不同租户具有不同的令牌，因此看到的资源是完全不一样的，即一个租户的用户无法访问另一租户的资源。OpenStack 中的用户在经过 Keystone 的验证后，会获得 Keystone 颁发并具有一定有效时长的 Token，在允许用户访问目标服务的 Endpoint 之前，服务会请求 Keystone 验证用户的 Token 是否有效，有效则允许访问相应的服务，否则拒绝访问，同时 OpenStack 中的每一个用户都会被分配一个与租户和用户相关的身份角色（Role）。图 2-8 为 OpenStack 项目架构中 Keystone 与各个服务之间的认证关系。

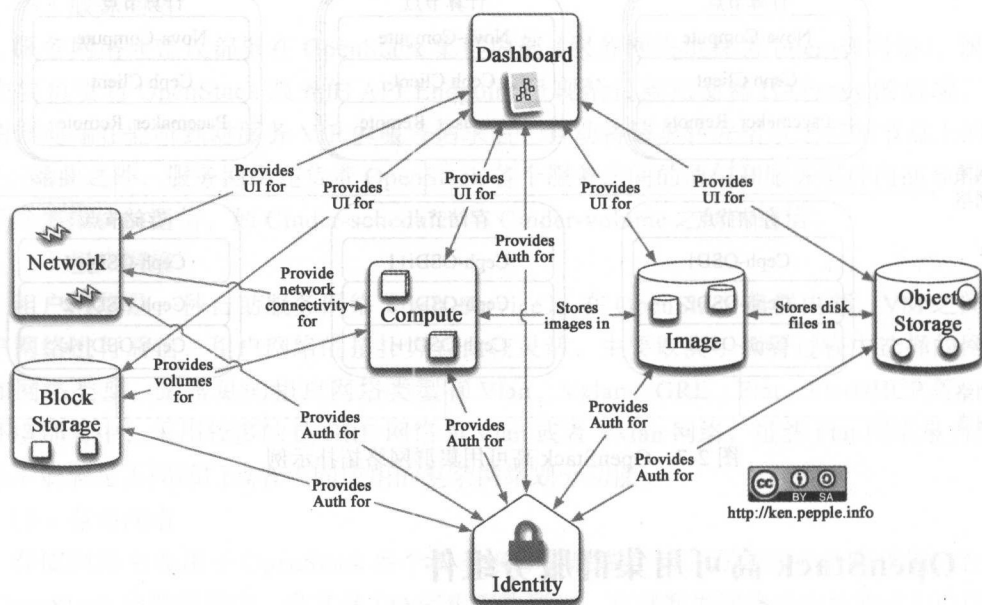


图 2-8 Keystone 在 OpenStack 架构中的认证服务

Keystone 的两个主要功能分别是用户管理和服务管理，要理解 Keystone 如何管理用户，则需要首先理解 Keystone 中用户管理的几个术语：

- ❑ User（用户）：User 是一个使用 OpenStack 云服务的人、系统或者服务的字符称号，Keystone 会对用户发起的资源调用请求进行验证，验证通过的用户可以登录 OpenStack 云平台并且通过 Keystone 颁发的 Token 去访问资源，用户可以被分配到一个或者多个 tenant/project 中。
- ❑ Credential（用户凭据）：Credential 是用来证明用户身份的数据，可以是用户名和密码、用户名和 API key，或者是 Keystone 认证后分配的 Token。
- ❑ Authentication（身份认证）：Authentication 是验证用户身份的一个过程。Keystone

服务通过检查用户的 Credential 来确定用户的身份，在第一次对用户进行认证时，用户使用用户名 / 密码或者用户名 / API key 作为 Credential，而当用户的 Credential 被验证后，Keystone 会给用户分配一个 Authentication Token 供该用户后续的请求使用。

- ❑ Token (令牌): Token 是一个 Keystone 分配的用于访问 OpenStack API 和资源服务的字符文本字符串。一个用户的 Token 可能在任何时间被撤销 (revoke)，即用户的 Token 是有时间限制的，在 OpenStack 中，Token 是和特定的 Tenant 绑定的，因此如果用户属多个 Tenant，则会有多个 Token。
- ❑ Tenant (租户): Tenant 是对资源进行分组或者隔离的容器 (有时也称为 Project)，一个 Tenant 可能对应一个云服务客户、一个服务账号、一个组织或者一个项目。在 OpenStack 中，用户可以属于多个 Tenant，并且必须至少属于某个 Tenant。Tenant 中可使用资源的限制称为 Tenant Quotas，它包括该 Tenant 内可使用的各种资源限额，如 CPU 限额、内存限额等。
- ❑ Service (服务): Service 是一个 OpenStack 提供的服务，比如 Nova、Swift 或者 Glance 等，在 OpenStack 中，每个服务提供一个或者多个 Endpoint 以供不同角色的用户进行资源访问和操作。
- ❑ Endpoint (服务入口): Endpoint 是一个 Service 监听服务请求的网络地址，客户端要访问某个 Service，则必须通过该 Service 提供的 Endpoint，Endpoint 通常是个可访问的 URL 地址，在 OpenStack 的多 Region 部署架构中，不同 Region 中相同的 Service 也具有不同的 Endpoint。在 OpenStack 服务框架中，Service 之间的相互访问也需要通过服务的 Endpoint 才可访问对应的目标 Service，例如当 Nova 服务需要访问 Glance 服务以获取 Image 时，Nova 就需要首先访问 Keystone，从 Keystone 的服务列表中拿到 Glance 的 Endpoint，然后通过访问该 Endpoint 以获取 Glance 服务。每个服务的 Endpoint 都具有 Region 属性，具有不同 Region 属性值的 Endpoint 也是不相同的。通常，一个服务会提供三类 Endpoint 供客户端使用：属性为 adminurl 的 Endpoint 只能被具有 admin 角色的用户访问；属性为 internalurl 的 Endpoint 被 OpenStack 内部服务用以彼此之间的服务通信；属性为 publicurl 的 Endpoint 用于其他用户的访问。
- ❑ Role (角色): Role 可以看成是一个访问控制列表 (ACL) 的集合。在 Keystone 的认证机制中，分配给用户的 Token 中包含了用户的角色列表。被用户访问的服务会解析用户角色列表中的角色所能进行的操作和可以访问的资源，在 Keystone 中，系统默认使用 admin 和 _member_role 角色。
- ❑ Policy (策略): Keystone 对用户的验证除了包含对用户的身份进行验证，还需要鉴别用户对某个服务是否有访问权限 (根据 Role 判断)。Policy 机制就是用来控制某一个 Tenant 中的某个 User 是否具有某个操作的权限。这个与相关的 Role 关联

的 User 能执行什么操作，不能执行什么操作，就是通过 Policy 机制来实现的。对于 Keystone 服务来说，Policy 就是一个 JSON 格式的文件，默认位置是 /etc/keystone/policy.json。通过配置这个文件，Keystone 认证服务实现了基于用户角色的权限管理。

在 Keystone 中，不仅只是用户与 Keystone 发生认证机制，OpenStack 内部各服务之间的通信也要 Keystone 的参与。以用户创建实例虚拟机为例，用户首先向 Keystone 发送如用户密码之类的身份信息，Keystone 验证成功后向用户配发 Token，之后用户向 Nova 发出带有 Token 的实例创建请求，Nova 接收到请求后向 Keystone 验证 Token 的有效性，Token 被证实有效后，由 Nova 向 Glance 服务发出带有 Token 的镜像传输请求，Glance 同样要到 Keystone 去验证 Token 的有效性，被证实有效后 Glance 向 Nova 正式提供镜像目录查询和传递服务，Nova 获取镜像后继续向 Neutron 发送带有 Token 的网络创建服务，再由 Neutron 向 Keystone 求证 Token 的有效性，Token 被证实有效后，Neutron 允许 Nova 使用网络服务，Nova 启动虚拟机成功，同时向用户返回创建实例成功的通知，图 2-9 为用户创建实例过程中 Keystone 的响应流程。

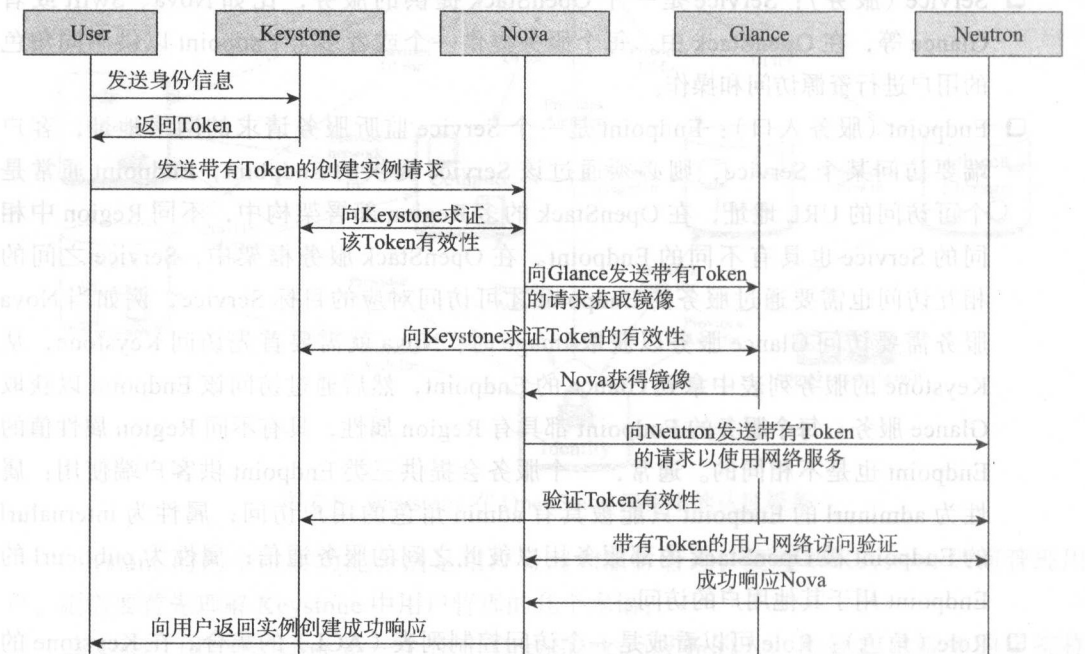


图 2-9 Keystone 实例创建认证过程

2.2.2 镜像服务 Glance

Glance 主要负责 Image 的注册和查询传送服务，Glance 中的镜像可以是用户制作并上传的镜像，也可以是对当前实例进行快照形式 copy 后的镜像副本，两种类型的镜像都可以快速用于实例部署。通过 Glance 提供的标准 RESTful API 接口，Glance 存储的镜像可以供

OpenStack 用户和管理员的多台服务器进行并行查询和访问，默认情况下，Glance 将用户上传的镜像存储在部署 Glance 服务的主机目录上（/var/lib/glance/images）。Glance API 服务可以通过配置缓存机制来提高镜像服务的响应速度，Glance 支持多种后端存储服务，例如本地文件系统作为存储介质、Swift（OpenStack Object Storage）作为存储介质或者 AWS S3 兼容的 API 作为存储介质以及分布式存储集群 Ceph 等。

Glance 作为 OpenStack 的一个核心系统项目，其架构是基于组件设计的，Glance 具有高可用、可容错及故障恢复、标准高度开放和兼容性强等特点。在 Glance 中需要理解的两个术语是镜像格式（Formats）和容器格式（Containers）。虚拟机镜像格式是底层的磁盘镜像格式，虚拟化供应商利用不同的磁盘镜像格式来存储虚拟机镜像数据，目前 Glance 支持的镜像格式包括：

- ❑ aki 镜像格式。aki 是 AWS 的 Kernel 镜像格式。
- ❑ ari 镜像格式。ari 是 AWS 的 Ramdisk 镜像格式。
- ❑ ami 镜像格式。ami 是 AWS 的虚拟机镜像格式。
- ❑ raw 镜像格式。raw 是一种非结构化的镜像格式，一个没有扩展过的镜像文件通常就是 raw 格式镜像。
- ❑ vhd 镜像格式。vhd 是一种通用的虚拟机磁盘格式，可用于 Vmware、Xen、Microsoft Hyper-V、VirtualBox 等虚拟化引擎中。
- ❑ vmdk 镜像格式。vmdk 是 Vmware 的虚拟机磁盘格式，支持多种 Hypervisor。
- ❑ vdi 镜像格式。vdi 是 VirtualBox、QEMU 等支持的虚拟机磁盘格式。
- ❑ iso 镜像格式。iso 是光驱数据的存档格式，如 CD-ROM。
- ❑ qcow2 镜像格式。qcow2 是一种支持 QEMU 并且支持动态磁盘镜像扩展和写时复制（Copy on Write）的磁盘格式。

容器格式主要用于表明虚拟机镜像文件是否包含实际虚拟机的元数据信息，容器类似一个文件夹，文件夹中可能包含磁盘镜像和实际虚机的元数据信息也可能只有镜像文件而没有元数据信息。目前使用较多的镜像容器格式有如下几种：

- ❑ docker 格式。一种 docker 容器格式。
- ❑ ovf 格式。一种开发式虚拟机磁盘格式，最初由 Vmware 发起，目前已被多种虚拟化设备支持。
- ❑ bare 格式。裸格式，这表示镜像没有 Container 或者没有包含元数据信息。
- ❑ aki 格式。Amazon Kernel 镜像。
- ❑ ari 格式。Amazon Ramdisk 镜像。
- ❑ ami 格式。Amazon 虚拟机镜像。



提示 当前 OpenStack 的镜像服务和其他项目还不支持容器格式，因此，在不确定的情况下，最好将镜像的容器设置为 bare 格式。

OpenStack 的镜像服务主要由 Glance-api 和 Glance-registry 两个服务构成。Glance 的 API 提供 V1 和 V2 两个版本，V1 版本的 API 只提供了基本的 Image 和 Member 操作功能，包括镜像创建、删除、下载、列表、详细信息查询、更新，以及镜像 Tenant 成员的创建、删除和列表，而 V2 版本的 API 还提供了镜像 Location 的添加、删除和修改等操作，以及 Metadata 的命名空间和 Image tag 等操作。目前 Glance 中的镜像数据分为两部分存放，Image 的元数据通过 Glance-registry 存放在数据库中，而 Image 的 Chunk 数据则通过 Glance-store 存放在各种 Backend Store（后端存储）中，并从中获取。Glance 的客户端主要是镜像的查询和使用者，包括 Horizon、Nova 和 Glance 的命令行，而后端存储的可选用范围也比较广泛，包括块存储 Cinder、本地文件系统、对象存储 Swift 和分布式存储 Ceph 等，图 2-10 是 Glance 的原理架构图。

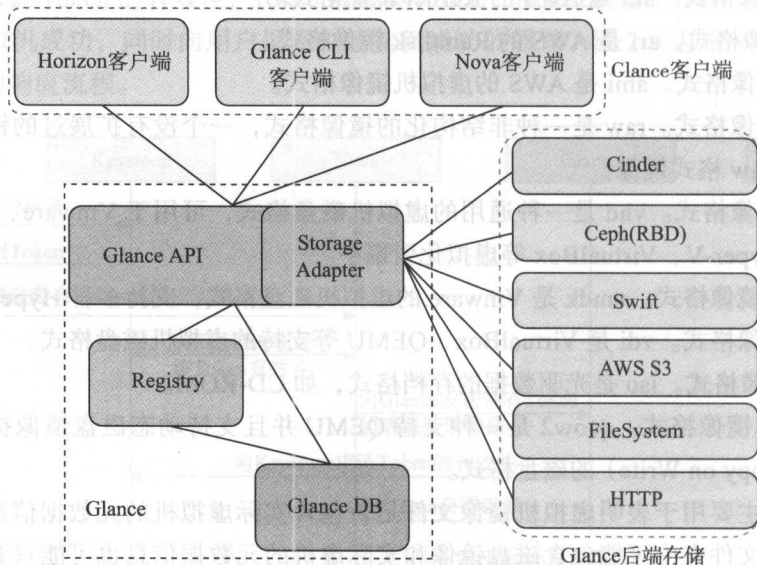


图 2-10 Glance 原理架构图

2.2.3 计算服务 Nova

Nova 是 OpenStack 中最为核心和复杂的项目之一，也是 OpenStack 中使用最多和最为成熟稳定的项目。Nova 的主要功能是对虚拟机进行管理并提供虚拟机运行所需的主要资源，Nova 是 OpenStack 作为 IaaS 服务的基石，此外，Nova 在通用服务器上进行计算能力弹性水平扩展的特性是 OpenStack 成为先进云计算平台最为主要的原因之一。在 OpenStack 提供的 IaaS 服务中，Nova 还提供了实例生命周期管理、计算资源管理、网络与授权管理、基于 REST 的 API 服务和异步连续通信等功能，虽然其中的某些功能模块已从 Nova 项目中分离成为独立项目，但是这些项目的基本功能仍然没变。在 OpenStack 中，Nova 与其多个 OpenStack 服务之间会进行相互访问，如 Nova 使用 Keystone 服务进行访问者的身份验证，

2.2.4 块存储服务 Cinder

表 2-1 Cinder 支持的后端存储矩阵

[illegible]

(续)

序号	驱动名称	创建卷	删除卷	卷挂载	卸载卷	卷扩展	创建快照	删除快照
50	Violin Memory	Kilo	Kilo	Kilo	Kilo	Kilo	Kilo	Kilo
51	VMware	Havana	Havana	Havana	Havana	Icehouse	Havana	Havana
52	Windows Server 2012	Grizzly	Grizzly	Grizzly	Grizzly		Grizzly	Grizzly
53	X-IO technologies	Kilo	Kilo	Kilo	Kilo	Kilo	Kilo	Kilo
54	Zadara Storage	Folsom	Folsom	Folsom	Folsom	Havana	Havana	Havana
55	Solaris (ZFS)	Diablo	Diablo	Diablo	Diablo			
56	ProphetStor (Flexvisor)	Juno	Juno	Juno	Juno	Juno	Juno	Juno
57	Infotrend	Liberty	Liberty	Liberty	Liberty	Liberty	Liberty	Liberty

各厂商存储设备在 Cinder 中的使用很简单，在厂商已经实现 Cinder 后端存储驱动的前提下，用户只需在 Cinder 的配置文件中将后端存储驱动进行相应的替换即可，图 2-12 是 EMC 块存储设备（VNX/VMAX 系列）在 Cinder 项目中的整合使用架构图。

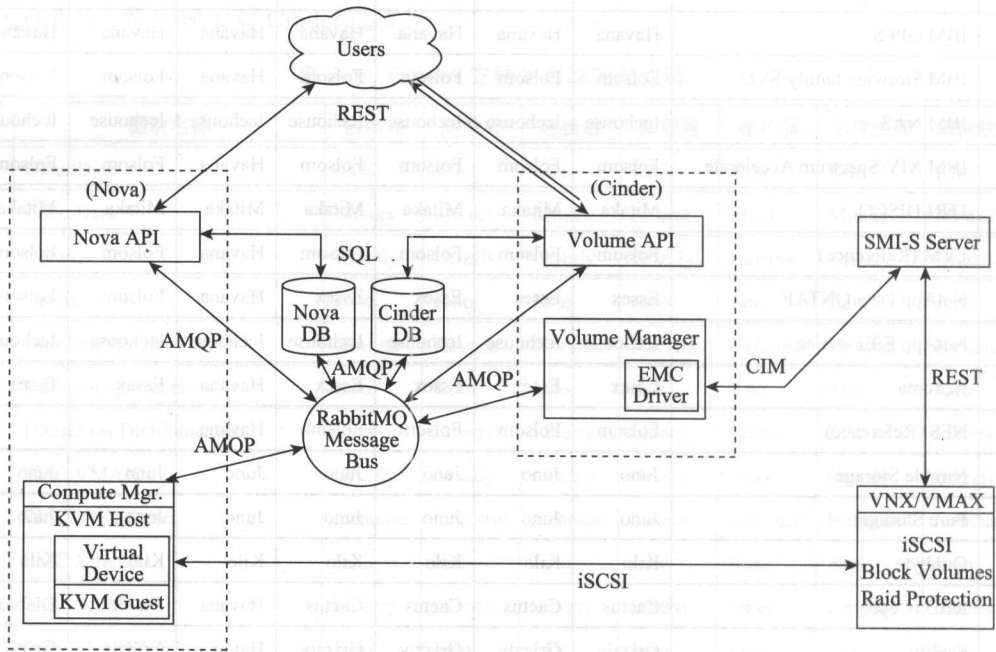


图 2-12 EMC 存储在 Cinder 中的使用

在 OpenStack 实例创建过程中，Nova-compute 服务仅提供虚拟机部署过程中 Profile 指定的镜像临时存储空间（即系统盘），而虚拟机的永久性数据块存储则由 Cinder 服务提供，如果 Nova 在创建实例时使用的是 SANBOOT 形式，则实例镜像存储和永久性块数据存储均由 Cinder 服务提供。在 Cinder 服务中，块存储可以像 SNA 网络存储一样在不同虚拟机

之间进行反复的 Attach 与 Detach 操作。随着越来越多厂商对 Cinder 的拥抱与支持，Cinder 项目所提供的功能越来越强大，而这些功能已经可以在很大程度上对企业级存储服务器的功能进行替换，如卷启动（SAN Boot）、卷复制（Volume Replication）、卷迁移（Volume Migration）和一致组（Consistency Group）等功能在 Cinder 中都已经实现。

2.2.5 网络服务 Neutron

Neutron 是 OpenStack 云平台中的网络服务，Neutron 的起源是 Nova 项目中的 Nova-network 服务，在 OpenStack 的 Folsom 版本中，Nova-network 被分离出来并成为了独立的网络服务项目 Quantum，由于商标冲突的原因 Quantum 在 Havana 版本中被改名为现在的 Neutron，因此，在不同版本的 OpenStack 部署架构中，网络服务组件可能会以 Nova-network、Quantum 和 Neutron 的名称出现，而它们都指代 OpenStack 的网络服务。Neutron 在 OpenStack 中提供网络即服务（Network as a Service, NaaS），其主要功能包括：提供面向租户的 API 接口，用于创建虚拟网络、路由器、负载均衡等，关联虚拟机实例到指定的网络和路由；通过 API 接口管理虚拟或物理交换机；提供 Plugin 架构来支持不同的网络技术平台；提供固定私网地址（Fixed IP）和公网浮动 IP 地址（Floating IP）。Neutron 向用户提供了网络自助服务的功能，用户可以按需对网络进行定义、隔离、加入网络等操作，同时 Neutron 还提供了多种灵活的网络模式以适应不同用户的网络环境。

同早期的 Nova-network 服务相比，Neutron 具有更加丰富的网络功能，Nova-network 只提供了一组静态的网络拓扑，其支持的网络模型主要是 Flat、FlatDHCP、VLAN 三种，并且 Nova-network 所提供的高级服务也只有 CloudPipe（VPNaaS）一种，而 Neutron 具有更丰富的网络拓扑结构和多层网络功能，同时 Neutron 还支持 VxLAN 和 GRE 等网络模型，其丰富的插件功能使得 Neutron 在网络设备商中得到普遍支持，Neutron 提供的高级功能除了 VPNaaS，还有 LBaaS、FWaaS 等，因此，尽管目前还有很多用户在使用 Nova-network，但是 OpenStack 社区已不再支持 Nova-network 的更新发展，在接下来的版本中，Nova-network 即将被抛弃。

在 Neutron 的功能组件中，最重要的两个组件就是 Neutron-Server 和插件。Neutron-Server 中包含守护进程 Neutron-server 和各种网络插件（以 Neutron-xxx-plugin 命名），Neutron-Server 既可以安装在控制节点也可以安装在网络节点，Neutron-server 提供网络访问的 API 接口，并把对 API 的调用请求转给已经配置好的 Plugin 进行后续处理，而插件需要访问数据库来维护各种虚拟网络设备的配置数据和对应关系，如路由器、网络、子网、端口、浮动 IP、安全组等与网络相关的数据。在 Neutron 中，最常见的两个 Agent 分别是 DHCP-Agent 和 L3-Agent，DHCP-Agent 主要负责向各个租户网络提供 IP 地址，L3-Agent 主要负责租户网络与外部网络的通信转发，图 2-13 为 Neutron 服务的内部组件交互架构图。

Neutron API 和插件构成了 Neutron-Server，Neutron 通过插件（Plugin）和插件代理（Plugin Agent）的组合来实现 Neutron API 转发来的网络服务请求，通常 Neutron API 和

Plugin 部署在网络节点或者控制节点上以接收和转发用户网络服务请求，而对应的 Plugin Agent 则主要部署到计算节点上以响应和实现用户的网络服务请求，Neutron 的内部功能实现如图 2-14 所示。

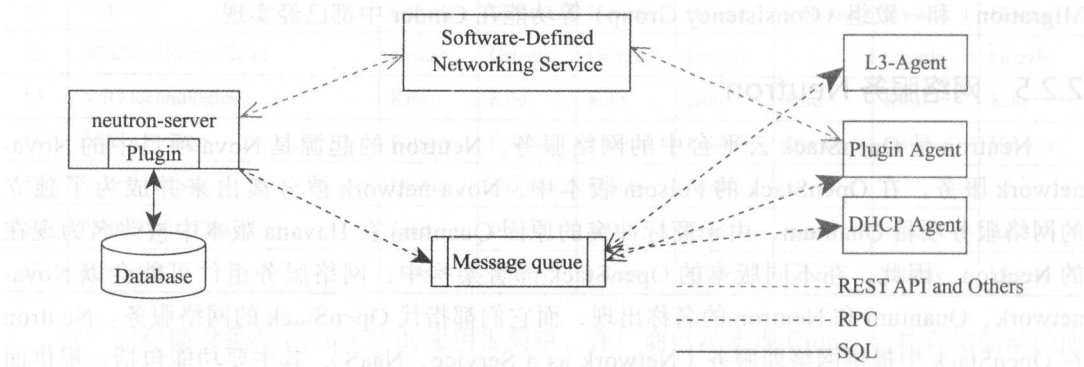


图 2-13 Neutron 组件交互架构图

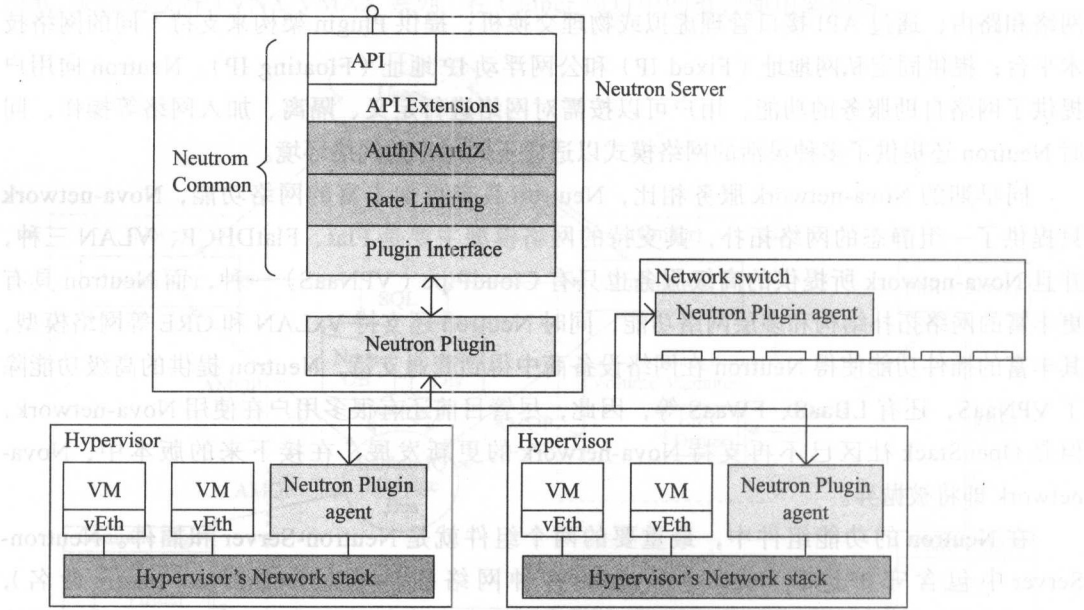


图 2-14 Neutron 功能实现架构图

通过以插件的形式来实现逻辑 API 的请求，使得各种不同的网络技术在 OpenStack 中都可以兼容使用，这对于希望避免厂商锁定和技术自主可控的用户而言，提供了极大的可选择空间，到 Mitaka 版本为止，Neutron 支持的插件如下：

- ❑ Open vSwitch Plugin。
- ❑ Cisco UCS/Nexus Plugin。
- ❑ Cisco Nexus1000v Plugin。

- ❑ Linux Bridge Plugin。
- ❑ Modular Layer 2 Plugin。
- ❑ Nicira Network Virtualization Platform (NVP) Plugin。
- ❑ Ryu OpenFlow Controller Plugin。
- ❑ NEC OpenFlow Plugin。
- ❑ Big Switch Controller Plugin。
- ❑ Cloudbase Hyper-V Plugin。
- ❑ MidoNet Plugin。
- ❑ Brocade Neutron Plugin Brocade Neutron Plugin。
- ❑ PLUMgrid Plugin。
- ❑ Mellanox Neutron Plugin Mellanox Neutron Plugin。
- ❑ Embrane Neutron Plugin。
- ❑ IBM SDN-VE Plugin。
- ❑ CPLANE NETWORKS CPLANE NETWORKS。
- ❑ Nuage Networks Plugin。
- ❑ OpenContrail OpenContrail Plugin。
- ❑ Lenovo Networking Lenovo Networking Plugin。

就普通用户而言,使用最多的还是 Modular Layer2 Plugin (ML2)、Open vSwitch Plugin (OVS) 和 Linux Bridge Plugin,其他与具体厂商相关的网络插件功能及其使用方法,可以参考 OpenStack 官网提供的插件介绍 (<https://wiki.OpenStack.org/wiki/Neutron>)。

2.2.6 控制面板 Horizon

Horizon 是 OpenStack 云平台的控制面板 (Dashboard), Dashboard 使得云管理员和用户能够通过 Web 界面进行云计算资源的创建、管理、监控等操作 (图 2-15)。Dashboard 使用基于 Web 的 API 接口与 OpenStack 云控制器进行交互,从而获取云资源的实时状态,在 OpenStack 自定义使用过程中,普遍会对 Horizon 进行二次开发,由于 Horizon 是基于 Django 框架的 Web 应用程序,其架构设计遵循一般 Web 架构,其在开发过程中主要涉及前端 UI 等内容,相比 OpenStack 的其他核心组件,Horizon 的开发难度相对较低,但是由于视觉效果的变化,往往使得用户更有兴趣对 Horizon 进行二次开发,另外对于 OpenStack 而言,凡是涉及新增 Web UI 显示内容,则通常都会涉及 Horizon 的开发集成工作。

Horizon 最初只是管理 OpenStack 计算项目 Compute 的简单应用程序,因此为了管理 Compute 项目的需要,最初的 Horizon 只有视图、模板和 API 调用三个功能。随后,Horizon 逐渐发展成为支持多个 OpenStack 组件和 API 的项目,并将这些功能严格划分到“dash”和“syspanel”组中,经过几年的社区发展,目前 Horizon 的设计和架构具有如下特点:

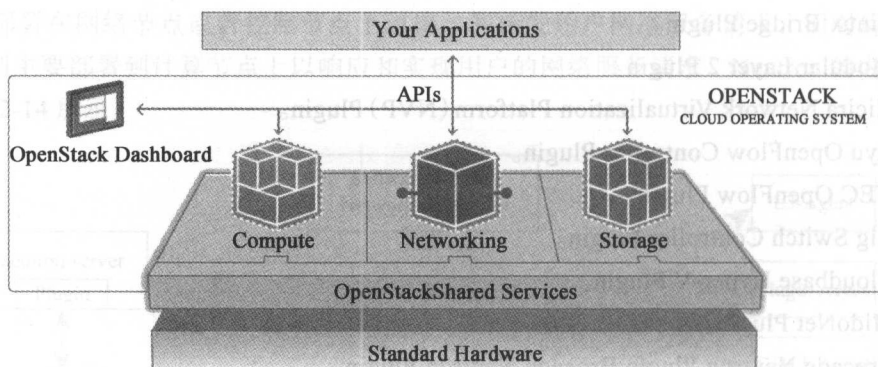


图 2-15 Horizon 与 OpenStack 核心项目的交互

- ❑ **Core Support** : 目前发行版本中的 Horizon 都包含三个核心的面板 (Dashboard), 分别是用户面板、系统面板和设置面板, 这三个 Dashboard 可以支撑几乎整个 OpenStack 的核心应用。为了向开发者提供一致、稳定和可重用的调用方法, Horizon 还提供了核心 OpenStack 项目的 API 抽象集合, 通过 Horizon 提供的抽象 API, 开发者无须透彻理解每个 OpenStack 项目的 API。
- ❑ **Extensible** : 在 Horizon 中, 任何人都可以添加一个组件到面板中。Horizon 的面板是基于 Dashboard 类开发的, Dashboard 类也被看成是 Horizon 的顶层导航条目。一个开发者可能要为自己的某个面板增加新功能 (例如在用户面板中增加监控功能), 那么开发者只需写一个 Django 的 App 并将其注册到 Dashboard 中即可, 因此, Horizon 具有很强的可扩展性。
- ❑ **Manageable** : 在 Horizon 中, 开发者可以轻松地将 Panel (导航条子项目) 注册到 Dashboard 中, 每一个 Panel 都包含了针对应用接口的必要逻辑 (视图、表格、测试等), 这种 Panel 级别的细粒度分解防止了成千上万行代码文件的出现, 并且只需将代码功能与导航条直接关联起来就可轻易定位代码。
- ❑ **Consistent** : Horizon 提供了核心类别来实现表格, 同时提供了可重复使用的固定模板集和其他的额外工具 (表格基类、滑动条基类、模板标签甚至基于类的视图), 用户可以很好地保证多个应用之间的一致性。
- ❑ **Stable** : 通过 Horizon 提供的核心类和组件的架构, Horizon 保证了用户对这些组件的修改将以向后兼容性的方式进行, 从而确保在整个 Horizon 生命周期中的稳定性。
- ❑ **Usable** : Horizon 向使用者提供了非常丰富的可用 API 接口, 因此开发者可以更多地关注自己感兴趣的功能开发, 而不是花费大量时间研究如何使用接口。

2.2.7 其他 OpenStack 服务

根据 OpenStack 官方的划分, OpenStack 核心项目主要有 6 个, 分别是: Nova、Neutron、

Cinder、Glance、Keystone 和 Swift，其余均为可选项目，但是这种划分并非对任何用户都适用，比如虽然 Swift 被作为核心项目，但是 Swift 的部署使用情况远没有 Horizon、Ceilometer 和 Heat 的部署使用率高，随着统一存储 Ceph 的出现，Swift 的使用普及率可能还会降低，因此对于大多数用户而言，为了满足云平台的核心功能，除了其他 5 个核心项目外，通常会部署使用的项目是控制面板服务 Horizon，而不是 Swift。随着 Liberty 版本的发行，OpenStack 社区也进入了 Big Ten 模式（“大帐篷”模式），几乎每隔一段时间就会有新的项目在孵化，而到 Mitaka 版本为止，OpenStack 社区接受和公布的项目就已经多达 20 个，这还不包括很多正在孵化的项目，因此，对于每个用户而言，将 OpenStack 的每一个项目全部进行部署显然是不切实际的。在 OpenStack 部署应用最多的项目中，除了前文介绍的几个项目外，以下几个项目的使用率或热门程度也比较高：

- ❑ Heat。Heat 是一种编排引擎，Heat 能够利用基于文本的模板文件同时部署多个复杂的云应用，用户在模板中写入量化的基础架构资源，这些资源可以包括实例模板配置参数、浮动 IP 地址、存储设备、安全组和用户等，之后用户便可使用此模板进行批量部署。
- ❑ Ceilometer。Ceilometer 是 OpenStack 的计量服务，Ceilometer 对 OpenStack 集群中的资源使用情况进行实时收集并汇总，尤其是在公有云中，Ceilometer 对于资源的监控和计量是非常重要的，因为这些参数可能是计费的参考标准。在 OpenStack 中，大多数服务项目都开发了 Ceilometer 的插件，Ceilometer 是一种中心化的资源参数收集服务，如要搜集某一服务的运行数据数据，只需开发一个与该服务相关的 Agent 即可。
- ❑ Swift。Swift 是一种高可用分布式对象存储，其在 OpenStack 各个项目中的成熟度和项目出现年限都是比较高的，只是由于概念和部署的复杂性以及功能的可替代性，使得其在用户环境中部署使用率一直较低。Swift 采用层次数据模型，共设三层逻辑结构：Account/Container/Object（即账户/容器/对象），每层节点数均没有限制，可以任意扩展。Swift 中的账户可理解为租户，用来做顶层的隔离机制；容器代表封装一组对象，类似文件夹或目录；底层的对象由元数据和内容两部分组成。Swift 由众多子服务组件构成，包括代理服务（Proxy Server）、认证服务（Authentication Server）、缓存服务（Cache Server）、账户服务（Account Server）、容器服务（Container Server）、对象服务（Object Server）、复制服务（Replicator）、更新服务（Updater）、审计服务（Auditor）、账户清理服务（Account Reaper）。由于 Swift 服务组件众多，设计相对复杂，在使用前需要投入大量精力研究以了解其架构原理。
- ❑ IroniC。IroniC 是 OpenStack 中的物理裸机管理服务，其在 OpenStack 中以完整项目发行出现是在 Kilo 版本中，因此其出现时间相对于其他项目是比较晚的。在目前的 OpenStack 中，虚拟化管理部分已经很成熟，用户通过 Nova 可以进行创建虚拟机、虚拟磁盘、管理电源状态、快速通过镜像启动虚拟机等操作，但是 OpenStack 在物

理机管理上一直没有成熟的解决方案，而 IroniC 的出现便是为了解决这一难题，虽然 IroniC 目前的成熟度不是很高，但是其在大规模物理服务器部署环境中仍然具有很好的前景。

❑ Magnum。Magnum 是 OpenStack 中与 Docker 集成的容器服务，可以说 Magnum 是随着 Docker 的火热而诞生的。近两年来，随着 Docker 的出现，业界关于 OpenStack 与 Docker 孰优孰劣的讨论一直不绝于耳，在这种情况下，OpenStack 社区主动拥抱容器技术，Magnum 项目应运而生，并成为 OpenStack 最热门的项目之一。截至 OpenStack 的 Mitaka 版本，Magnum 可以为用户提供 Kubernetes-as-a-Service、Swarm-as-aService 和 Mesos-as-a-Service 服务，用户可以很方便地通过 Magnum 来管理 Kubernetes、Swarm 和 Mesos 集群，通过 Magnum 和后台的 COE (Container Orchestration Engine，包括 Kubernetes、Swarm 和 Mesos) 来交互获取容器服务。随着容器技术的持续火热，Magnum 项目聚集了大量开发者，是目前社区极为活跃的项目。

到目前为止，OpenStack 社区最为成熟的核心项目也还未满七年，而新项目也层出不穷。根据 OpenStack 官方社区的调查，在目前社区接受的项目中，最为成熟和被普遍使用的仍然还是以计算、存储、网络为主的核心服务，截至目前，OpenStack 社区各个项目的成熟度和使用情况如表 2-2 所示。

表 2-2 OpenStack 项目现状概要

项目名称	服务名称	成熟度	项目年限	部署率
Nova	Compute	8 of 8	5 Yrs	93%
Neutron	Networking	8 of 8	4Yrs	84%
Cinder	Blok Storage	7 of 8	4Yrs	81%
Glance	Image Service	7 of 8	5 Yrs	87%
Keystone	Identity	7 of 8	5 Yrs	88%
Swift	Object storage	7 of 8	5 Yrs	52%
Horizon	Dashboard	6 of 8	5 Yrs	86%
Manila	Shared Filesystems	5 of 8	2 Yrs	11%
Heat	Orchestration	4 of 8	4 Yrs	64%
Trove	Database	3 of 8	2 Yrs	17%
Sahara	Elastic Map Reduce	3 of 8	2 Yrs	11%
IroniC	Bare-Metal Provisioning	3 of 8	2 Yrs	20%
Ceilometer	Telemetry	2 of 8	4 Yrs	59%
Zaqar	Messaging Service	2 of 8	2 Yrs	2%
Murano	Application Catalog	2 of 8	2 Yrs	15%

(续)

项目名称	服务名称	成熟度	项目年限	部署率
Designate	DNS Service	1 of 8	2 Yrs	17%
Barbican	Key Management	1 of 8	2 Yrs	2%
Magnum	Containers	1 of 8	1 Yrs	11%
Congress	Governance	0 of 8	1 Yrs	1%

2.3 Redhat OpenStack 高可用部署架构

Redhat 与 Marientis 均是 OpenStack 社区代码贡献最为强劲的两家厂商，同时各自都有自己基于 OpenStack 的产品线，Redhat 的 OpenStack Platform (OSP) 系列产品，Marientis 的 Fule 系列产品都在 OpenStack 个人和企业部署市场上占有较大的份额。本节主要介绍 Redhat 的 OpenStack 高可用部署方案。

2.3.1 Redhat OpenStack 高可用集群部署架构

Redhat 的 OpenStack 高可用方案主要分为两种方式：一种是基于 Keepalived 的高可用方案（图 2-16 所示）；一种是基于 HAproxy 负载均衡器和 Pacemaker 集群资源的高可用方案^①。后一种方案又可以分为服务独立的集群高可用部署（HACluster-deployment-segregated）方案（图 2-17 所示）和服务集中的集群高可用部署（HACluster-deployment-collapsed）方案（图 2-18 所示）。由于后一种方案利用了 Pacemaker 强大的资源管理功能，因此，Redhat 官方推荐的部署方案是基于 HAProxy 和 Pacemaker 的高可用部署方案。

HAproxy 是一种基于 TCP/IP 的负载均衡器，在 Redhat 架构中，HAproxy 用于对 Openstack 的访问请求进行负载均衡，HAproxy 支持 Active/Active 模式，Active/Passive 模式等多种负载均衡模式。Pacemaker 是一种开源的高可用资源管理器，它可以检测到服务器级别和应用级别的故障并能够将服务从故障中恢复，在 Redhat 高可用架构中，如果 OpenStack 的某个服务故障，则 Pacemaker 负责重启该服务，同时 Pacemaker 还负责 OpenStack 使用的基础组件的高可用，如数据库高可用、Message 队列和 HAproxy 负载均衡器的高可用。图 2-17 和图 2-18 中描述了每个 OpenStack 服务的高可用模式，这是 Redhat 高可用集群经常使用的服务高可用模式，这些高可用模式包括：

- Active/Active 模式。也称为双活模式，当服务配置为主 / 主模式时，全部节点都同时运行相同服务并同时接受访问请求，如果其中一个节点故障，则该节点的访问请求被转移到其他任一正常节点，整个过程客户端的服务请求不会中断。
- Active/Passive 模式。也称为主备模式，当服务配置为此主 / 备模式时，只有一个节

① Beekhof. Scripts for deploying a HA install of OSP [CP/OL]. Github.com.<https://github.com/beekhof/osp-ha-deploy>.

点运行服务并接受请求，当此节点故障时，备节点服务会被激活，后续的请求会被转发到备节点上，整个过程中客户端请求会有短暂中断。

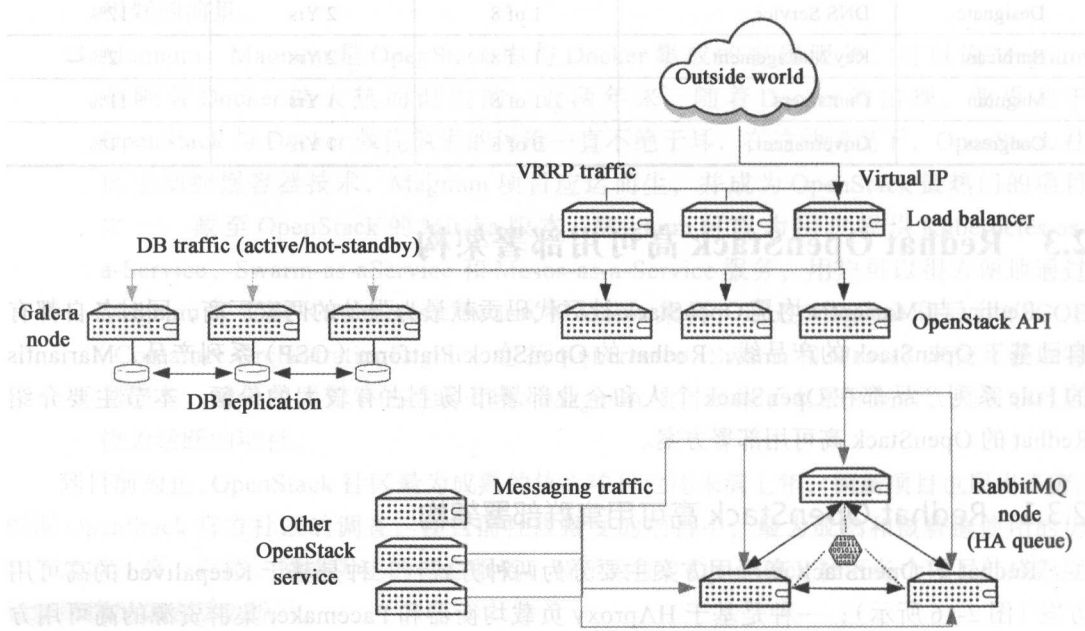


图 2-16 Redhat 基于 Keepalived 的 OpenStack 高可用集群部署方案

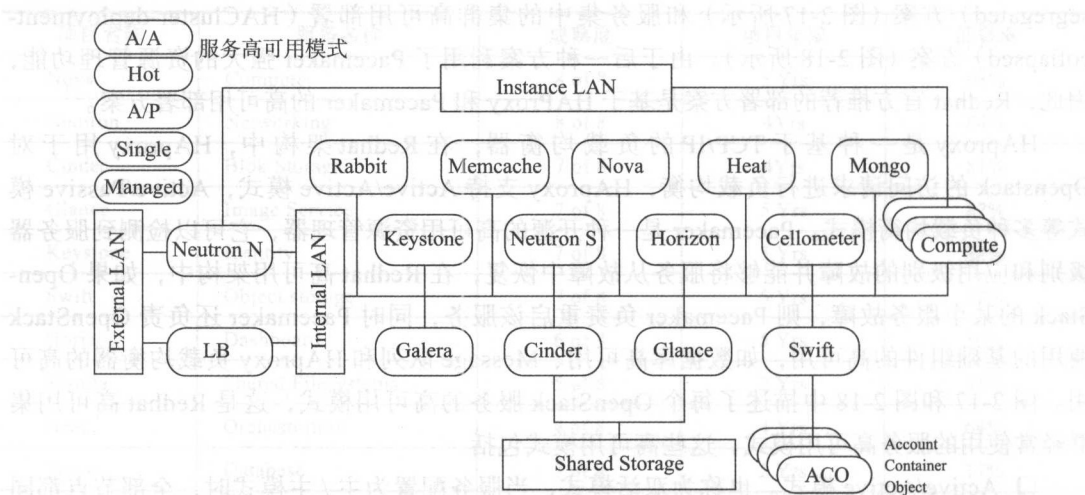


图 2-17 Redhat 服务独立的 OpenStack 高可用集群部署方案

❑ Hot Standby 模式。也称为热备模式，当服务配置为热备模式时，所有节点都同时运行相同服务，但是只有一个节点在响应客户端的访问请求，其余节点没有客户端的请求访问，当接受请求的节点故障时，客户端访问请求被无缝切换至运行相同服务的正常节点上继续处理，整个过程客户端的服务请求不会中断。

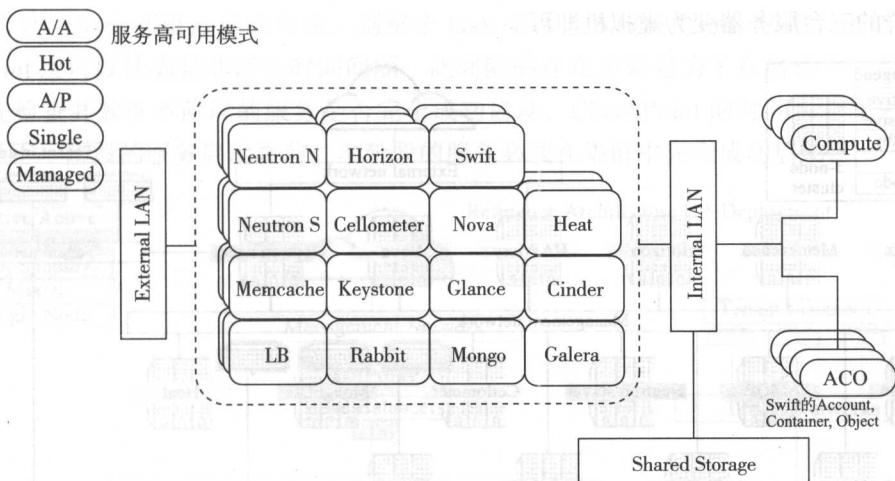


图 2-18 Redhat 服务集中的 OpenStack 高可用集群部署方案

- ❑ Mixed 模式。混合模式通常用在由多个子服务组成的服务组件中，即将服务组件的子服务配置成上述几种高可用模式组合，混合模式的配置因服务情况而异，通常，混合高可用模式指某些子服务配置成为主 / 主模式，而有些配置成为主 / 备或者热备模式。
- ❑ Single Node 模式。也称为单点模式，即服务仅在一个节点运行，这类服务可以通过 Pacemaker 来管理，如果 Pacemaker 检测到其故障则重新启动它，整个过程会有相对较长时间的中断。

为了实现每一个服务的独立和高可用，Redhat 从概念上设计了一个理想的高可用服务集群部署方案，在这种理想的高可用集群部署模式下，每一个服务都被分散部署在独立的物理服务器（或者虚拟机）上，并且运行每个 OpenStack 服务组件的一组服务器（Redhat 推荐三个服务器为一组）组成一个高可用集群，最终实现独立的集群负责独立服务组件的高可用性。在这种概念设计的高可用模式下，每个 OpenStack 服务组件均运行在一组独立的服务器上，同时服务器的数量还可以根据服务的负载集群进行按需扩容，图 2-19 即为这种理想化的 OpenStack 高可用集群概念设计方案^①。

当然，这种理想化的 OpenStack 高可用集群方案要变为现实，对于很多用户而言是很难接受的，因为为了保证每个服务的独立和高可用，在不考虑计算节点的情况下就需要使用大量服务器来提供各种 API 服务，更重要的是，对于 OpenStack 的服务而言，最消耗物理资源（或者说最需物理资源）的往往是计算服务，而不是各种 API 和调度服务。当然，如果用户确实需要将各个服务组件隔离到独立的集群下运行，又不希望浪费大量的物理服务器，Redhat 建议可以将各个 OpenStack 服务隔离到虚拟机上运行，即只需将图 2-19 中每个

① Alex Baretto. Deployment A highly available OpenStack [J/OL]. Redhat Enterprise OpenStack Platform Witepaper. 2013. <https://access.redhat.com/articles/1370143>.

服务集群的三台服务器变为虚拟机即可。

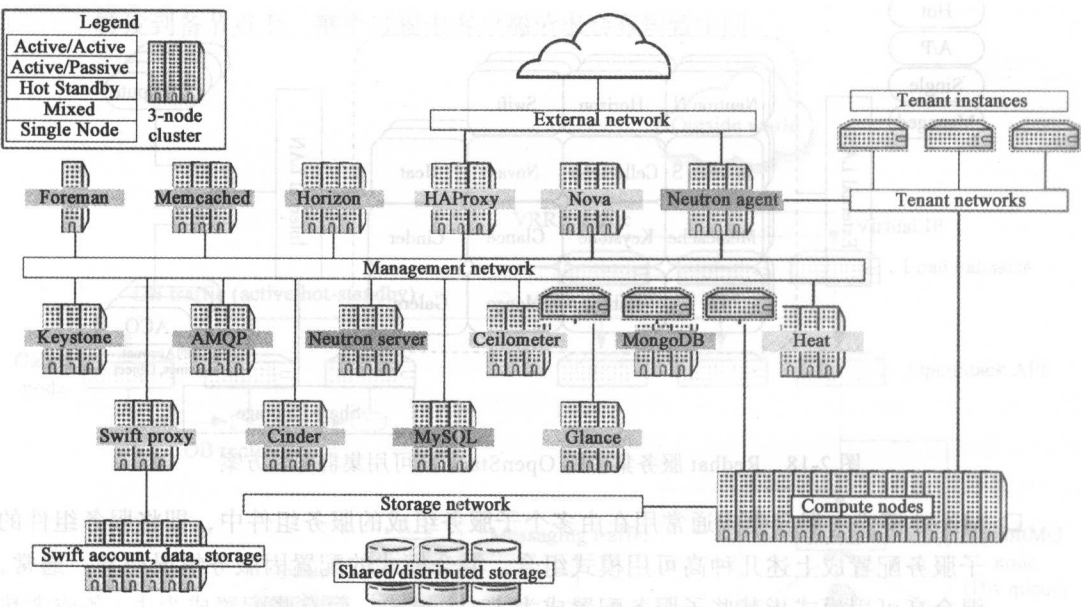


图 2-19 Redhat 概念化的 OpenStack 高可用集群方案

在实际部署中，Redhat 也提供了服务集中式的高可用部署方案，即将 OpenStack 的控制、API、调度等服务集中到控制节点集群（控制节点集群至少由三台服务器组成）上运行，同时将计算服务独立到计算节点上运行，OpenStack 运行在控制节点集群上的服务通过 Pacemaker 集群管理软件进行高可用管理，计算节点通过精简的 Pacemaker_Remoter 服务以远程节点的形式加入控制节点集群中，通过控制节点的集群管理软件统一实现高可用管理，图 2-20 为 Redhat 提供的服务集中式 OpenStack 高可用集群部署方案。

在 OpenStack 的集群服务中，各服务之间是一种相互依赖和交互的关系，即某些服务的存在是以另一服务的存在为前提的，而某些服务之间可能并不存在相互依赖关系，所以这部分服务是可以独立存在或者并行启动的，集群服务之间的这种依赖关系涉及了服务启动的先后顺序，当然由于资源管理器 Pacemaker 的存在，这种启动顺序和依赖关系可以得到很好解决。用户要注意的是如何区分哪些服务是可以并行启动的，哪些服务之间因为有依赖关系而必须是先后启动的，Redhat 为这种 OpenStack 服务的依赖关系也提供了参考，图 2-21 便是 Redhat 在 OpenStack 高可用集群中推荐的服务启动依赖关系图。

图 2-21 描述了 Redhat 在 OpenStack 高可用集群中分配的各个服务之间的启动依赖关系，从中可以看到，整个集群最先启动的一定是最底层的负载均衡器，然后是支撑 OpenStack 服务的基础类服务如消息队列 AMQP、数据库 MariaDB（或者 MySQL）和 MongoDB 紧随其后可以并行启动，接着是认证服务 Keystone 和 Memcached 可以同时启动，之后是 OpenStack 核心服务，如 Nova、Neutron、Cidner 等跟着可以并行启动，最后，

一旦 IaaS 所需的全部服务启动完成, 则整个 IaaS 平台进入服务 Ready 状态。图 2-21 中的 CheckPoint 层可以认为是非活动时间间隔, 此间隔的存在主要是为了在启动下一阶段的服务之前先检查并验证本阶段的服务是否完全成功启动, CheckPoint 的时间大小不一, 关键是要保证下一阶段的服务启动之前, 本阶段的服务必须在集群中完全成功启动。

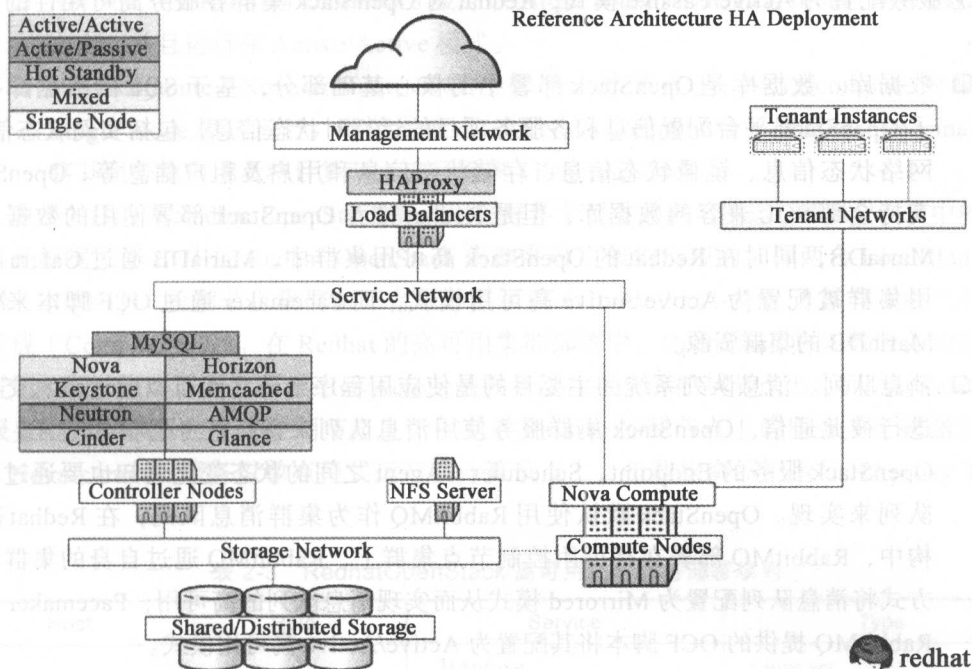


图 2-20 Redhat 实际应用的 OpenStack 高可用集群部署架构

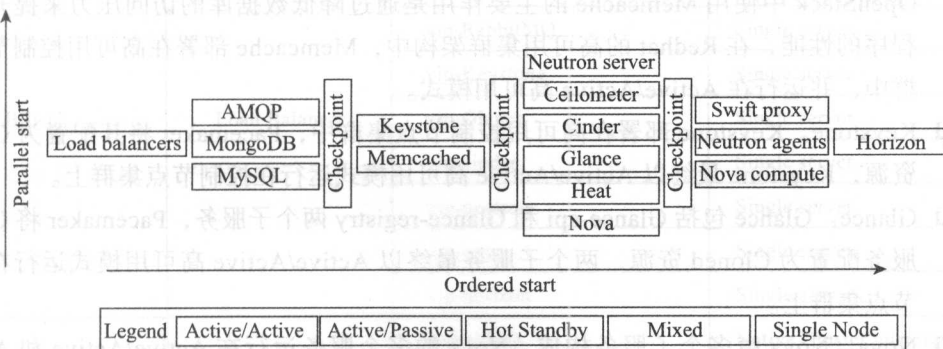


图 2-21 Redhat OpenStack 高可用集群服务启动依赖关系

2.3.2 Redhat OpenStack 高可用集群服务规划

Redhat 的 OpenStack 高可用集群部署方案推荐使用的是 HAProxy 和 Pacemaker 来进行高可用和资源管理, Redhat 为每个 OpenStack 集群服务分配虚拟 IP (VIP), VIP 均衡分布

到三台控制器集群上，并通过 Pacemaker 来进行 HAProxy 的高可用，HAProxy 将客户端对 VIP 的访问转发到具体的后端 OpenStack 集群服务中进行处理，OpenStack 集群服务的高可用性通过 Pacemaker 进行控制。由于集群服务特性各自不一，因此，Redhat 将不同的 OpenStack 集群服务配置成为不同的高可用模式，如无状态服务配置为 Active/Active 模式，有状态服务配置为 Active/Passive 模式，Redhat 对 OpenStack 集群各服务高可用性的配置如下：

- ❑ 数据库。数据库是 OpenStack 部署中的核心基础部分，基于 SQL 的数据库存储了 OpenStack 平台配置信息和各服务项目的运行时状态信息，包括实例状态信息、网络状态信息、镜像状态信息、存储状态信息和用户及租户信息等。OpenStack 支持全部 SQL 兼容的数据库，但是 Redhat 推荐 OpenStack 部署使用的数据库为 MariaDB，同时在 Redhat 的 OpenStack 高可用集群中，MariaDB 通过 Galera 高可用集群被配置为 Active/Active 高可用模式，而 Pacemaker 通过 OCF 脚本来管理 MariaDB 的集群资源。
- ❑ 消息队列。消息队列系统的主要目的是使应用程序组件之间可以通过消息交换来进行彼此通信，OpenStack 集群服务使用消息队列来进行任务通信和交互，同时 OpenStack 服务的 Endpoint、Scheduler、Agent 之间的状态变化通知也要通过消息队列来实现。OpenStack 默认使用 RabbitMQ 作为集群消息队列，在 Redhat 的架构中，RabbitMQ 部署在高可用控制节点集群上，RabbitMQ 通过自身的集群配置方式将消息队列配置为 Mirrored 模式从而实现消息队列的高可用，Pacemaker 通过 RabbitMQ 提供的 OCF 脚本将其配置为 Active/Active 高可用模式。
- ❑ Memcache。Memcache 是将 Key-Value 数据类型存储在内存中的小型数据库。OpenStack 中使用 Memcache 的主要作用是通过降低数据库的访问压力来提升应用程序的性能，在 Redhat 的高可用集群架构中，Memcache 部署在高可用控制节点集群中，并运行在 Active/Active 高可用模式。
- ❑ Keystone。Keystone 部署在高可用控制节点集群中，Pacemaker 将其配置为 Cloned 资源，Keystone 最终以 Active/Active 高可用模式运行在控制节点集群上。
- ❑ Glance。Glance 包括 Glance-api 和 Glance-registry 两个子服务，Pacemaker 将 Glance 服务配置为 Cloned 资源，两个子服务最终以 Active/Active 高可用模式运行在控制节点集群上。
- ❑ Nova。Nova 由多个子服务构成，Nova 的多个服务运行在 Active/Active 和 Active/Passive 组成的 Mixed 模式下，其中，Nova-api、Nova-scheduler、Nova-consoleauth 和 Nova-novncproxy 均部署在高可用控制节点集群上，Nova-api、Nova-consoleauth 和 Nova-novncproxy 均以 Cloned 资源的形式运行在 Active/Active 模式，Nova-scheduler 运行在 Active/Passive 模式。
- ❑ Cinder。Cinder 的多个服务也运行在 Mixed 模式，其中 Cinder-api、Cinder-scheduler

运行在 Active/Active 模式，Cinder-volume 运行在 Active/Passive 模式。

- ❑ Neutron。Neutron 由多个子服务构成，其中 Neutron-server、L3-agent、DHCP-agent、OpenVSwitch-agent、metadata-agent 均部署在高可用集群控制节点上，同时运行在 Active/Passive 模式。
- ❑ Horizon。Horizon 的 Dashboard 服务以 Httpd 资源的形式部署在高可用集群控制节点上，并且运行在 Active/Active 模式。
- ❑ Nova-compute。Nova-compute 部署在每个计算节点上，Nova-compute 的服务以 Single Node 模式运行，即单点模式，当 Nova-compute 服务故障时，Pacemaker 会将其服务重启或者将计算节点进行隔离（Fencing）操作。

表 2-3 归纳了 Redhat OpenStack 高可用集群各个服务的高可用运行模式^①，表中的服务部署是根据图 2-20 中的 Redhat OpenStack 集群高可用架构实现的，即两台 LoadBlancer 服务器（LB[0-1]），控制节点集群由三台服务器组成（Controller[2-4]），计算节点由三台服务器组成（Compute[5-7]）。在 Redhat 的高可用集群部署中，OpenStack 的多数服务均运行在高可用控制节点集群上，即 Active/Active 的服务运行在全部控制节点上，而 Active/Passive 的服务也需要部署在全部控制节点上，但是在 Passive 控制节点上的服务并未被激活，此外，Nova-compute 服务以单点模式运行在计算节点上，负载均衡器上运行 HAproxy 和 VIP 服务。

表 2-3 RedhatOpenStack 高可用集群服务部署规划

Host	Role	Service	Type
LB[0-1]	Load Balancer	HAproxy	Clone set
		vip-MariaDB	Single server
		vip-RabbitMQ	Single server
		vip-Keystone	Single server
		vip-Glance	Single server
		vip-cinder	Single server
		vip-neutron	Single server
		vip-nova	Single server
		vip-horizon	Single server
Controller[2-4]	controller	RabbitMQ	Native Cluster mirrored
		MariaDB	MariaDB-wsrepGalera Cluster
		Keystone	Clone set

① Jacob Liberman. High availability with Red Hat Enterprise Linux OpenStack Platform 4 [J/OL]. Red Hat Reference Architecture Series. 2014. <https://www.redhat.com/zh/resources/high-availability-red-hat-enterprise-linux-OpenStack-platform-4>.

(续)

Host	Role	Service	Type
Controller[2-4]	controller	Glance-api	Cloneset
		Glance-registry	Cloneset
		Cinder-api	Cloneset
		Cinder-scheduler	Cloneset
		Cinder-volume	Cloneset
		Openvswitch	Single server
		Neutron-server	Single server
		Neutron-openvswitch-agent	Single server
		Neutron-dhcp-agent	Single server
		Neutron-L3-agent	Single server
		Neutron-metadata-agent	Single server
		Nova-consoleauth	Clone set
		Nova-novncproxy	Clone set
		Nova-api	Clone set
		Nova-scheduler	Single server
Compute[5-7]	Compute node	Nova-conductor	Clone set
		Httpd	Clone set
		Openvswitch	Single server
		Nova-compute	Single server
		Libvirttd	Single server
		Messagebus	Single server
		Neutron-openvswitch-agent	Single server

OpenStack 高可用集群由诸多服务组件构成，为了保证每一个服务的高可用性和可扩展性，OpenStack 的高可用架构要求每一个服务都应该具备 Scale-out 的扩展特性，即在服务访问负载较大的情况下，可以实现服务处理能力的弹性扩展并能够将访问请求以负载均衡的形式分布到不同的服务节点上进行响应。为此，部署有 OpenStack 服务的每个控制节点都要求能够在不影响当前服务的情况下实现 Scale-out 扩展，而在集群中的某个控制节点出现故障的情况下 OpenStack 的服务可用性也不能受到影响。为了实现 OpenStack 服务的高可用性，Redhat 的高可用集群架构将 OpenStack 的每一个服务都交由集群资源管理器 Pacemaker 来管理，并通过 HAproxy 将访问负载均衡分布到每个集群处理节点上，同时为了防止集群脑裂 (Brain-split)，建议 OpenStack 控制节点集群至少由三台服务器构成。Redhat 推荐的 OpenStack 高可用部署架构下控制节点 Pacemaker 管理的高可用服务分布如

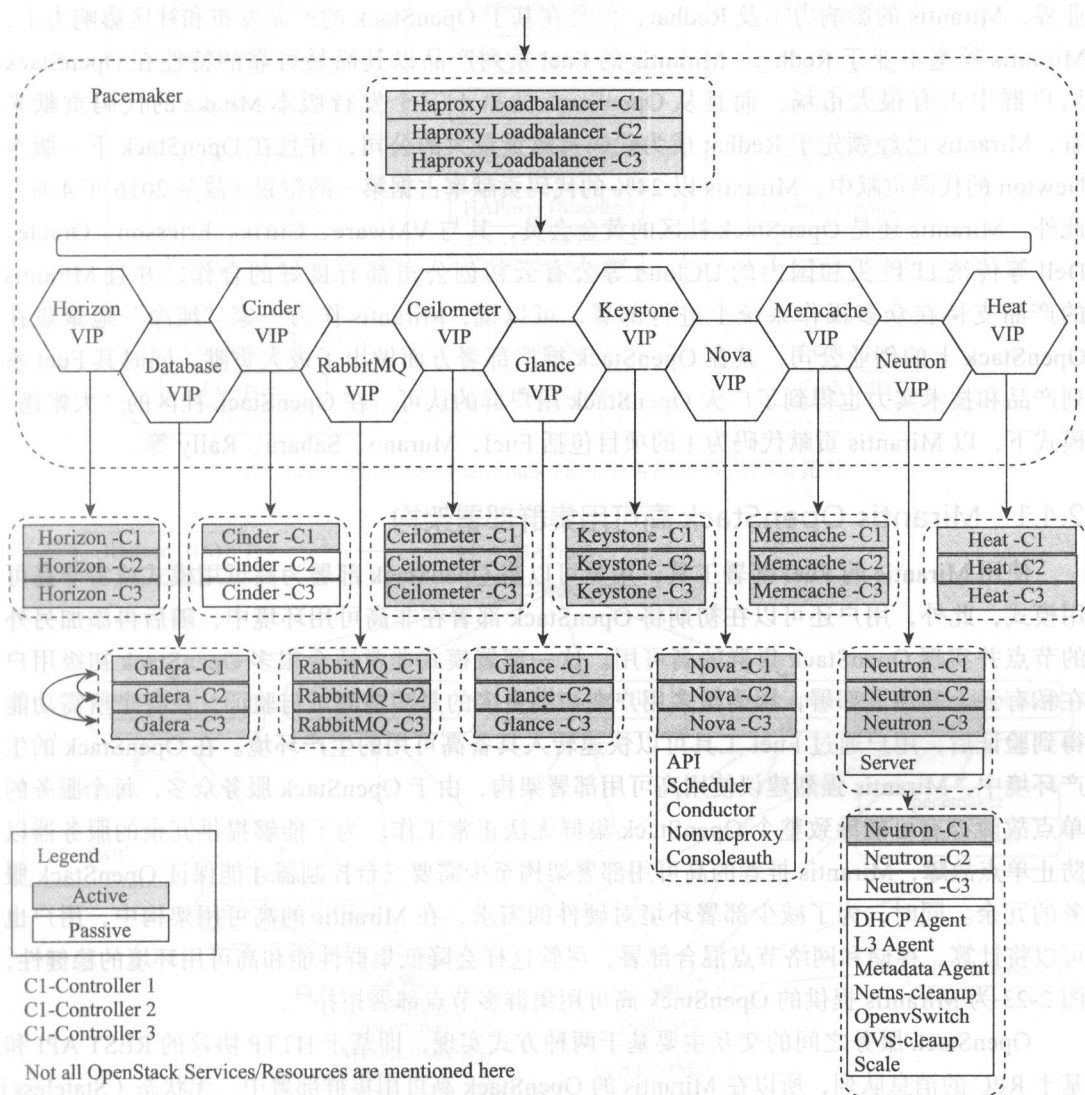
图 2-22 所示^①。

图 2-22 Redhat OpenStack 集群 Pacemaker 高可用服务

2.4 Mirantis OpenStack 高可用部署架构

Mirantis 是 OpenStack 最为成功的创业公司之一，其在 2013 年 10 月开始发布自己的

① Balaji Jayavelu. Deploying Highly Available Red Hat Enterprise Linux OpenStack Platform 6 with Ceph Storage [J/OL]. Red Hat Reference Architecture Series. 2015. <https://www.redhat.com/zh/resources/deploy-rhel-OpenStack-platform-6-with-ceph-storage>.

OpenStack 版本 Fuel, Fuel 也是基于 OpenStack 社区的发行版本。也许在开源社区或者 IT 业界, Mirantis 的影响力不及 Redhat, 但是在基于 OpenStack 的产品发布和社区影响力上, Mirantis 丝毫不亚于 Redhat。Mirantis 的 Fuel 系列产品以其简易可靠的特性在 OpenStack 用户群中占有很大市场, 而且从 OpenStack 的第十三个发行版本 Mitaka 的代码贡献来看, Mirantis 已经领先于 Redhat 成为代码贡献量最大的公司, 并且在 OpenStack 下一版本 Newton 的代码贡献中, Mirantis 以 24% 的代码贡献率占据第一的位置(截至 2016 年 4 月), 此外, Mirantis 还是 OpenStack 社区的黄金会员, 其与 VMware、Citrix、Ericsson、Oracle、Dell 等传统 IT 巨头和国内的 UCloud 等公有云初创公司都有良好的合作, 并且 Mirantis 的产品支持在众多操作系统上进行部署, 可以说, Mirantis 作为一家“纯净”地聚焦在 OpenStack 上的创业公司, 其在 OpenStack 推广部署方面做出了极大贡献, 同时其 Fuel 系列产品和技术实力也得到了广大 OpenStack 用户群的认可。在 OpenStack 社区的“大帐篷”模式下, 以 Mirantis 贡献代码为主的项目包括 Fuel、Murano、Sahara、Rally 等。

2.4.1 Mirantis OpenStack 高可用集群部署架构

使用 Mirantis 的 Fuel 部署工具, 用户可以将 OpenStack 部署为高可用模式或者非高可用模式, 此外, 用户还可以在初期将 OpenStack 部署在非高可用环境中, 随后再添加另外的节点并实现 OpenStack 集群的高可用。这一部署模式非常适合很多 OpenStack 初级用户在私有云上的递进部署, 因为很多用户在初期更多的是功能测试与验证, 在各个所需功能得到验证后, 用户通过 Fuel 工具可以快速转入具备高可用的生产环境。在 OpenStack 的生产环境中, Mirantis 强烈建议使用高可用部署架构, 由于 OpenStack 服务众多, 每个服务的单点故障都有可能整个 OpenStack 集群无法正常工作。为了能够提供冗余的服务器以防止单点故障, Mirantis 推荐的高可用部署架构至少需要三台控制器才能保证 OpenStack 服务的冗余。同时, 为了减少部署环境对硬件的需求, 在 Mirantis 的高可用架构中, 用户也可以将计算、存储和网络节点混合部署, 尽管这样会降低集群性能和高可用环境的稳健性, 图 2-23 为 Mirantis 提供的 OpenStack 高可用集群多节点部署拓扑^①。

OpenStack 服务之间的交互主要基于两种方式实现, 即基于 HTTP 协议的 REST API 和基于 RPC 的消息队列, 所以在 Mirantis 的 OpenStack 高可用集群部署中, 无状态(Stateless)服务如 OpenStack API 服务, 其高可用的实现主要借助 Pacemaker 管理的虚拟 IP 和负载均衡器 HAProxy 以实现客户端服务请求的负载均衡。而对于有状态的 OpenStack 服务组件, 诸如数据库和消息队列服务, 其高可用性则是通过这些组件自身的 Active/Active 或 Active/Passive 高可用模式来实现, 例如消息队列服务 RabbitMQ 使用其内置的集群机制实现消息队列的镜像高可用, 而数据库则使用 MySQL/Galera 复制机制来实现高可用, 图 2-24 为 Mirantis 提供的 OpenStack 高可用集群组件之间的交互与调用关系拓扑。

① Mirantis. Mirantis OpenStack 7.0 Reference Architecture [J/OL]. Mirantis. 2015. <https://docs.mirantis.com/OpenStack/fuel/fuel-7.0/reference-architecture.html>.

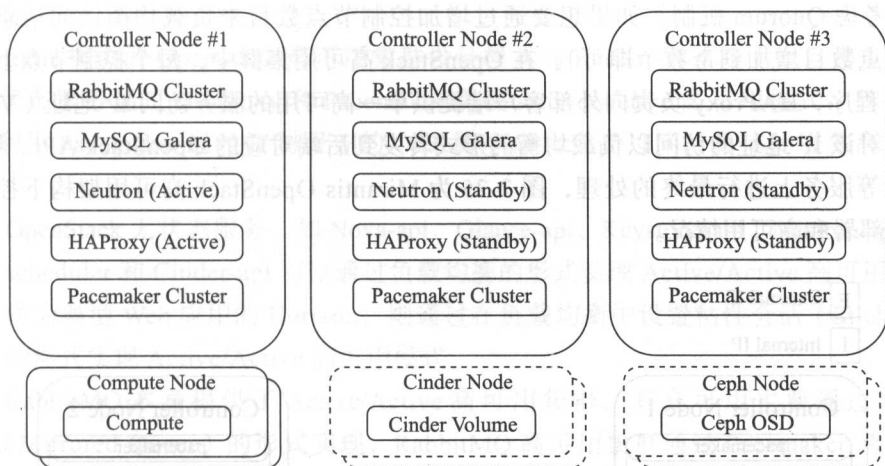


图 2-23 Mirantis OpenStack 高可用集群多节点部署拓扑

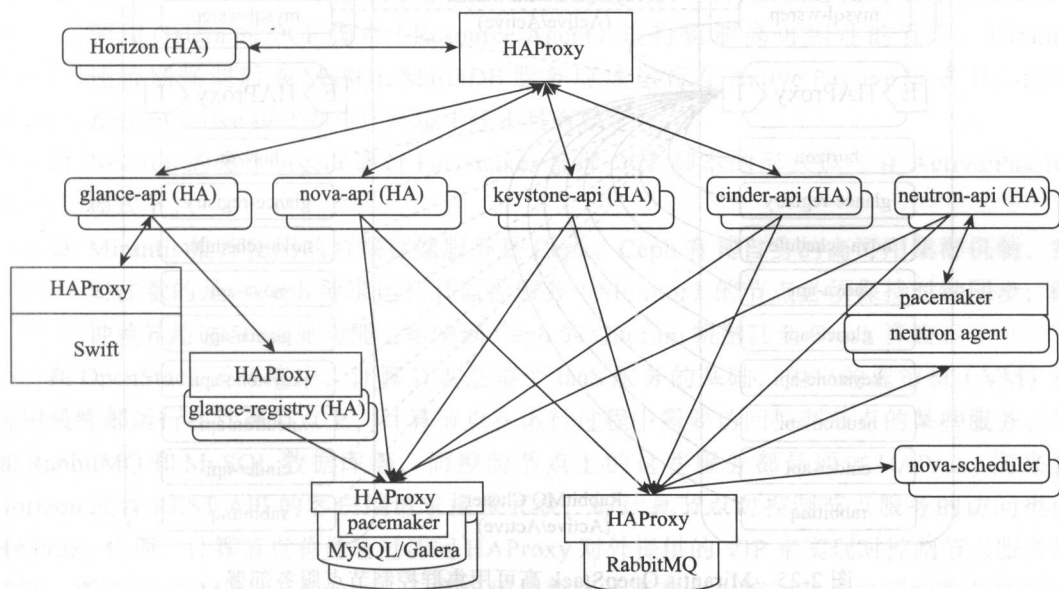


图 2-24 Mirantis OpenStack 高可用集群服务组件交互调用关系

Mirantis 多节点高可用 OpenStack 环境涉及三种类型的节点，分别是控制节点、计算节点、存储节点。业务系统高可用性设计的首要考虑便是物理设备的冗余，因此作为运行 OpenStack 服务最多的控制节点，必须实现服务器节点的冗余。由于 MySQL 数据库运行在控制节点上，并通过 Galera 来实现高可用，而 Galera 是一个基于 Quorum 的系统，因此作为高可用 OpenStack 集群的控制节点理论上至少需要 3 台服务器组成，而 Mirantis 的部署文档认为，除了集群服务可用性有所降低之外，两个控制节点也是可行的，而且控制节点和服务都可以通过 Scale-out 的方式进行扩容，因此后期也可以将控制节点增加到 5 个或

者 7 个（考虑 Quorum 机制，如果希望通过增加控制节点数目来负载均衡访问请求，只需将控制节点数目增加到奇数个即可）。在 OpenStack 高可用集群中，每个控制节点上均运行 HAProxy 程序，HAProxy 负责向外部客户端提供单一高可用的服务访问 IP 地址（VIP），并将客户端对该 IP 地址的访问以负载均衡的形式转发到后端对应的 OpenStack API、数据库、消息队列等服务上进行最终的处理，图 2-25 为 Mirantis OpenStack 高可用架构下控制节点上的服务部署和高可用情况。

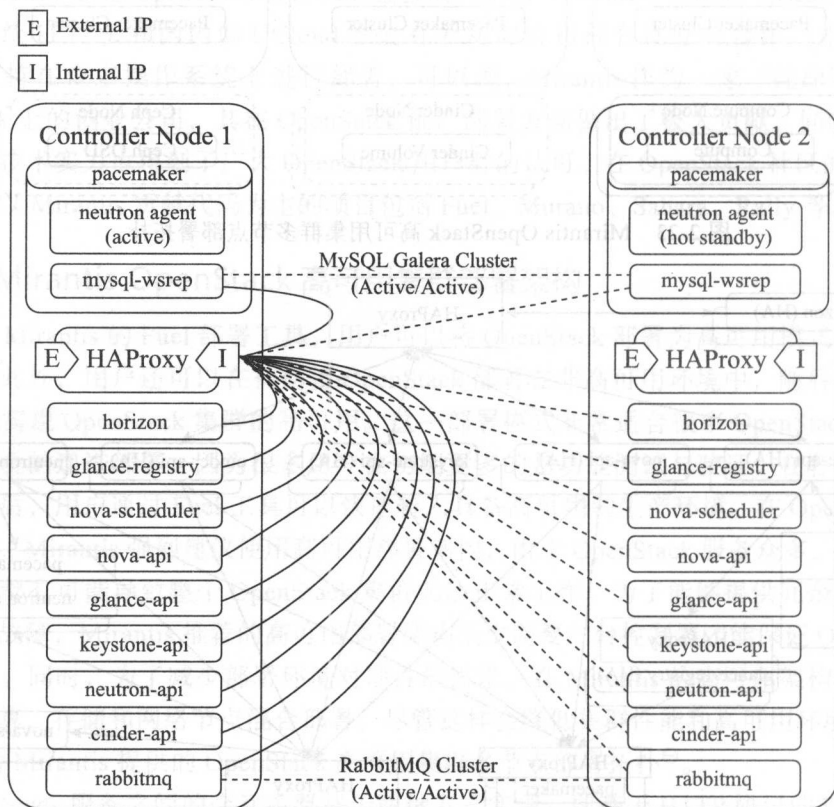


图 2-25 Mirantis OpenStack 高可用集群控制节点服务部署

在多节点 OpenStack 高可用环境下，当终端用户从 Dashboard 访问 OpenStack 云服务或者以命令行形式向 Openstack 服务，如 Nova-api、Glance-api、Keystone-api、Neutron-api、Nova-scheduler 或者 MySQL 服务的 REST API 发出访问请求时，客户端请求将会进入 HAProxy 活动的控制节点，因为 HAProxy 对外提供的服务 IP 地址（VIP）只会在 HAProxy 活动的节点上出现，同时客户端请求被 HAProxy 终止，即客户端请求是不会直接抵达 OpenStack 服务的 API 的，当客户端发起下一个访问请求时，HAProxy 仍然会将其终止并重新转发给位于后端控制节点上的相应 OpenStack 服务，而这个控制节点可能与之前处理请求的控制节点相同，也可能是另外的控制节点，主要取决于用户对 HAProxy 负载均衡策

略的设置。

Mirantis 提供的 OpenStack 高可用集群设计模式与 Redhat 差异并不大，其主要思想也是将无状态服务通过 HAProxy 和 Pacemaker 实现 Active/Active 高可用模式，其他有状态服务则利用 Pacemaker 通过控制服务的 OCF 脚本来实现高可用，具体的服务高可用模式如下：

- OpenStack 无状态服务，如 Nova-api、Glance-api、Keystone-api、Neutron-api、Nova-scheduler 和 Cinder-api 可以通过负载均衡的形式实现 Active/Active 高可用模式。
- 作为典型 Web 应用的 Horizon，则通过在负载均衡中设置粘性会话（Stick Session）的形式实现 Active/Active 高可用模式。
- RabbitMQ 本身提供了 Active/Active 高可用集群，其高可用主要通过队列镜像（Mirrored Queue）的形式实现，RabbitMQ 高可用集群通过 Pacemaker 调用客户自定义的 OCF 脚本（Resource Agent）来实现。
- MySQL/MariaDB 的高可用通过 Galera 集群实现，具体实现过程则通过 Pacemaker 调用 Galera 的 OCF 脚本（Resource Agent）进行资源高可用性的管理。Mirantis 还特别强调后端 MySQL/MariaDB 服务应该运行在 Active/Passive 模式下，因为 Active/Active 模式在生产环境中还不具有稳定性。
- Neutron 的多个 Agent 通过 Pacemaker 管理 OCF 脚本的形式运行在 Active/Passive 模式下。
- Mirantis 推荐使用的后端存储服务是 Ceph，Ceph 有其自身的高可用集群机制，需要注意的是，Ceph 要求运行其监控服务（Monitor）的节点必须保持时钟同步，时钟差异超过 50ms 则可能会影响到 Ceph 的 Quorum 机制甚至 Crush 算法。

在 OpenStack 云环境中，计算节点是整个 IaaS 服务的基础，用户的虚拟机（VM）和应用最终都运行在计算节点上，计算节点在运行过程中需要访问控制节点的某些服务，例如 RabbitMQ 和 MySQL 数据库等，而控制节点上的这些服务都是通过 HAProxy 向来自 Horizon 或者 REST API 的客户端请求提供冗余，即计算节点对控制节点服务的访问也由 HAProxy 代理，计算节点总是通过访问 HAProxy 对外提供的 VIP 来实现对控制节点服务的访问，图 2-26 是 Mirantis OpenStack 高可用集群下计算节点与控制节点之间的服务访问示意图。

在 Mirantis 的高可用架构下，存储节点可以有多种选择，例如用户可以在存储节点上部署使用 Cinder、Swift 或者 Ceph 来作为 OpenStack 集群的存储服务，从集群共享存储和高可用角度考虑，Mirantis 建议在存储节点上部署 Ceph 服务，用户需要注意的是保证足够的控制节点数目（通常为三个）

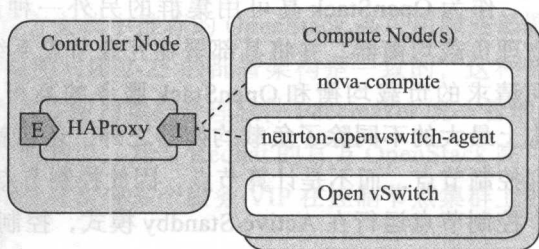


图 2-26 Mirantis OpenStack 高可用环境下计算节点与控制节点服务访问

以确保运行在控制节点上的 Ceph 监控服务 Monitor 能够实现 Quorum 机制，同时需要部署足够的 OSD 节点以满足 Ceph 数据的多份复制从而实现 Ceph 集群数据的冗余高可用，图 2-27 为 Mirantis 推荐的基于 Ceph 集群的存储节点拓扑图。

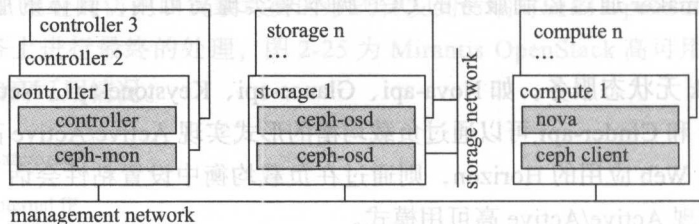


图 2-27 Mirantis OpenStack 高可用集群中的存储节点拓扑图

2.4.2 Mirantis OpenStack 自定义高可用集群架构

在 Mirantis 看来，OpenStack 生产环境部署最重要的两个方面是服务的高可用性和可扩展性，在满足这两个必要条件的前提下，OpenStack 服务在节点上的分布可以灵活多变。传统的 OpenStack 生产环境部署认为高可用环境就应该存在服务集中的控制节点（OpenStack APIs、MySQL/MariaDB 和 RabbitMQ 均部署在控制节点），而 Mirantis 认为这些 OpenStack 相关的服务均可以部署在控制节点或计算节点上，并将控制节点与计算节点在“胖节点”（部署大量服务）与“瘦节点”（部署少量服务）之间权衡，如果将控制节点上的服务移到计算节点，则控制节点甚至可以消失。图 2-28 中，计算节点作为“胖节点”部署了 Nova-api、Nova-scheduler、Glance-api 等服务，而控制节点作为“瘦节点”仅部署数据库和消息队列服务，同时使用物理负载均衡器来作为 OpenStack 服务的访问入口（Endpoint），将来自 Horizon 和 REST API 的访问请求进行均衡高可用。在这种部署模式下，控制节点仅部署平台服务（数据库和消息队列），其他属于 OpenStack 的内部服务全部部署在计算节点，因此用户可以将数据库独立出来交由专门的 DBA 团队负责，而消息队列也可独立到特定的消息集群中。通过这种方式，最终的 OpenStack 集群只剩下部署有“纯净”OpenStack 服务的计算节点，中心化的控制节点已经消失，不仅减轻了云管理的复杂性，同时可以对集群实现几乎是水平线性的伸缩扩展。

作为 OpenStack 高可用集群的另外一种部署方式，Mirantis 使用 HAProxy 软件替换物理负载均衡器，并将其部署在冗余节点系统上实现高可用，集群通过 HAProxy 实现访问请求的负载均衡和 OpenStack 服务的高可用性，如图 2-29 所示。图 2-29 与图 2-28 架构上最大的不同除了负载均衡器之外，还有就是图 2-29 中 OpenStack 的 APIs 服务被部署到控制节点，而不是计算节点，因此控制节点变得“更胖”，而计算节点变得“更瘦”，两个控制节点运行在 Active/Standby 模式，控制节点服务之间的故障转移通过 Pacemaker 和 Corosync/Heartbeat 实现。

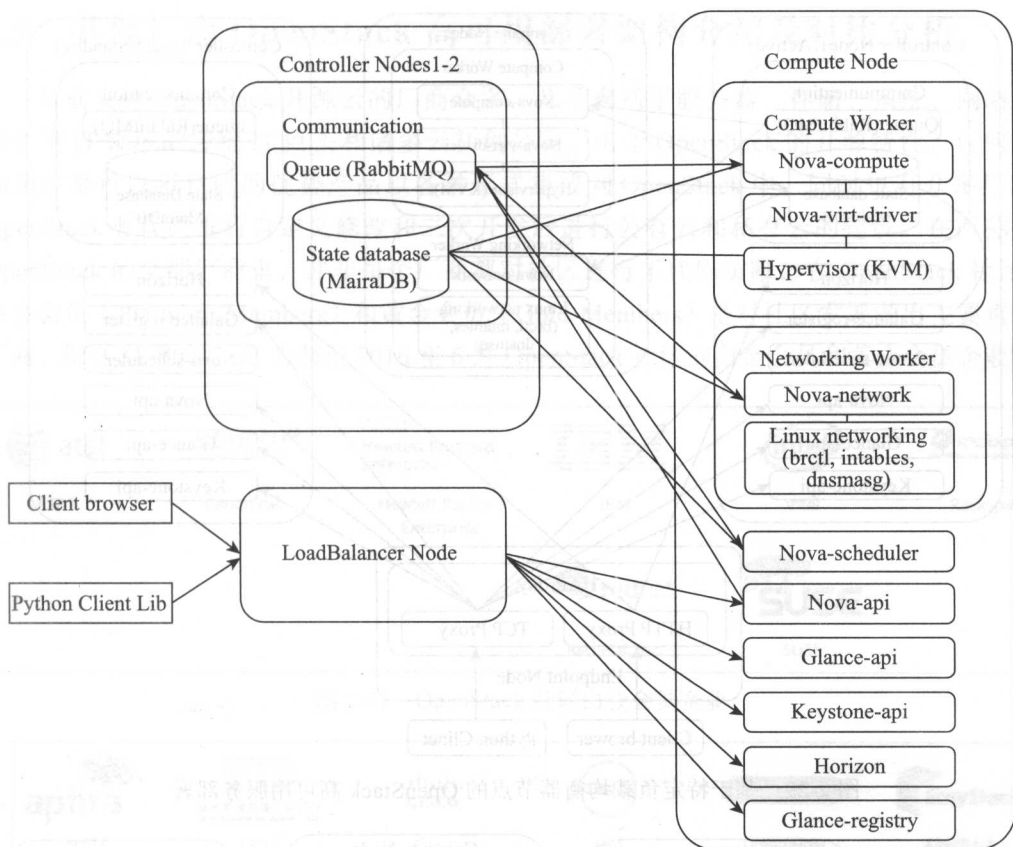


图 2-28 基于物理负载均衡器的 OpenStack 高可用服务部署

用户不仅可以通过物理负载均衡器和独立节点系统上的 HAProxy 软件实现 OpenStack 服务的负载均衡和高可用，还可以将 HAProxy 集成到控制节点，并将除了计算服务之外的 OpenStack 服务和基础平台服务全部部署在控制节点上，同时控制节点可以通过添加节点和重新配置 HAProxy 的形式实现伸缩扩展，如图 2-30 所示。在这种模式下，至少需要两个 HAProxy 实例以 Active/Standby 模式运行在控制节点上，HAProxy 的故障检测和某个 Standby 状态的 HAProxy 实例变为 Active 状态则通过 Pacemaker 和 Corosync/Heartbeat 实现。图 2-30 的高可用部署模式也是 Mirantis 官方文档中推荐的 OpenStack 高可用服务部署模式，即图 2-30 中的 OpenStack 高可用部署架构与图 2-25 的部署架构是一致的，这种部署模式的优点是将集群服务集中部署到控制节点并使用集群管理软件 Pacemaker 统一管理，同时集群所需物理设备和服务器节点数目也有所减少，其与 Redhat 的官方 OpenStack 高可用部署架构比较相似，但是 Mirantis 的架构并没有实现多个服务 VIP 在控制节点集群上的均衡分布，而是采用单一的 VIP 对外提供服务。

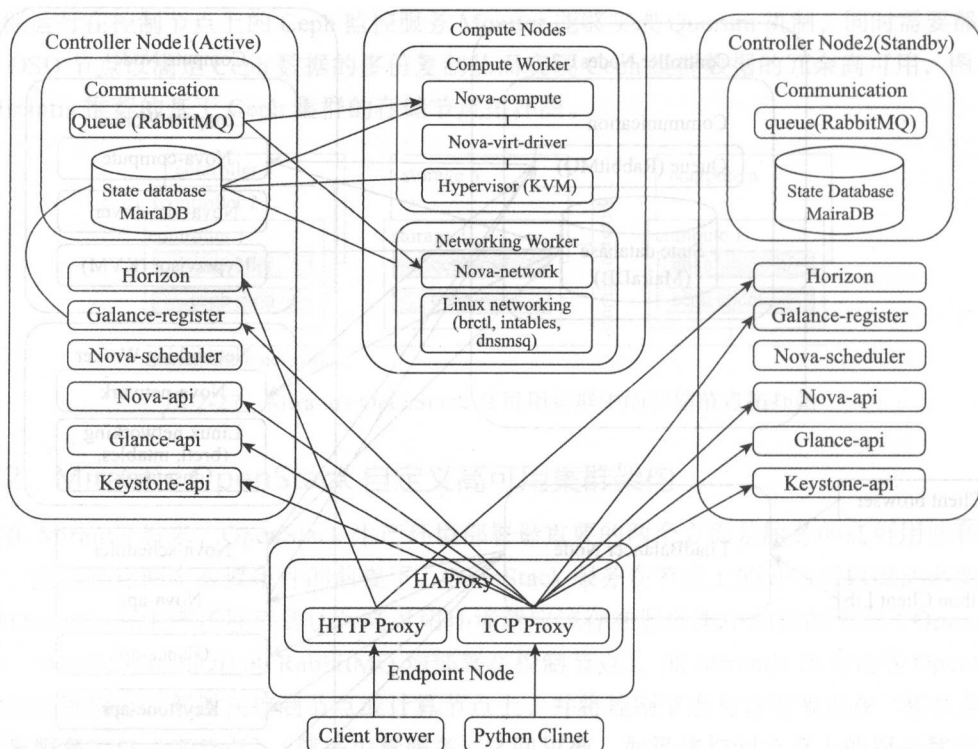


图 2-29 基于特定负载均衡器节点的 OpenStack 高可用服务部署

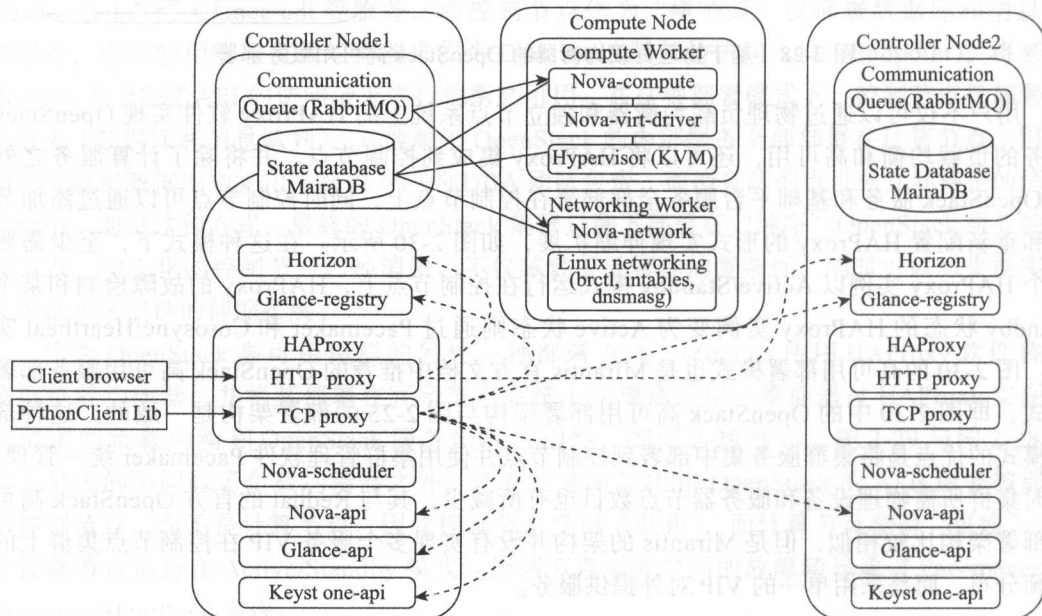


图 2-30 基于集成负载均衡器的 OpenStack 高可用服务部署

2.5 其他厂商 OpenStack 高可用部署架构介绍及对比分析

目前支持 OpenStack 开源云的厂商众多，几乎囊括了服务器、存储、系统、网络、虚拟化等 IT 领域的传统 IT 巨头和诸多云初创公司。由于 OpenStack 的开源特性，任何厂商和用户都可以将自己的优势产品以各种形式集成到 OpenStack 中，同时也有众多厂商对 OpenStack 源代码进行自定义修改和二次开发后进行公有云和私有云的建设。在声称拥护 OpenStack 的全部厂商里，并非每个厂商都对社区进行了代码贡献，在 OpenStack 社区中，白金会员（Platinum Members）和黄金会员（Gold Members）是对社区发展做出主要贡献的厂商，图 2-31 和图 2-32 为截至 2016 年 6 月 OpenStack 社区的白金会员和黄金会员企业^①。

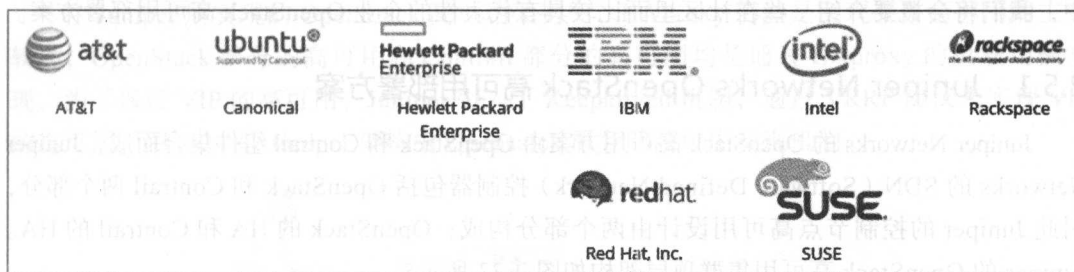


图 2-31 OpenStack 社区白金会员企业

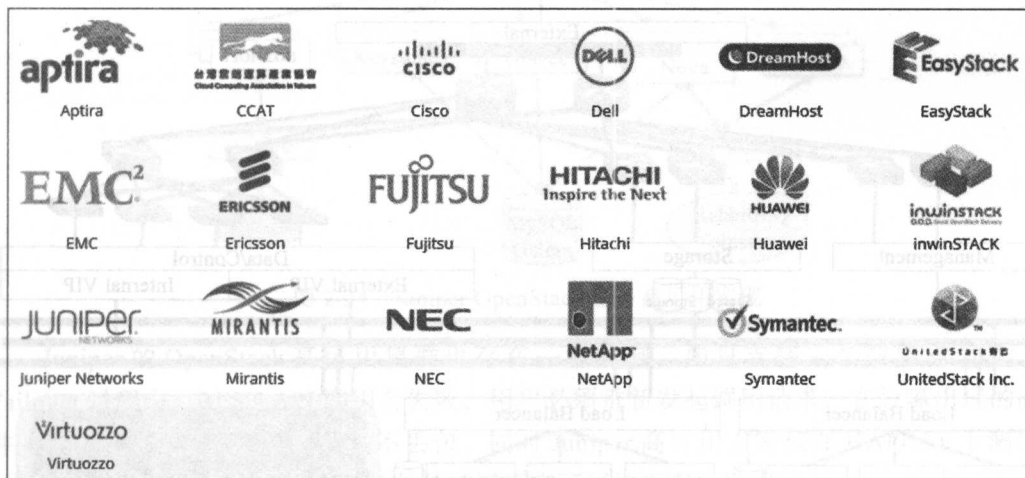


图 2-32 OpenStack 社区黄金会员企业

除了白金会员与黄金会员，OpenStack 社区庞大的企业赞助商（Corporate Sponsor）也是社区代码的主要贡献者和部署使用的实践者，诸如国内的华为、99Cloud、AWCloud、ZTE 和国外的 PayPal、Oracle、HPE、TCPCloud 等。在 OpenStack 的高可用部署实践与

① OpenStack. org. Companies Supporting The OpenStack Foundation [EB/OL]. OpenStack. org. <http://www.OpenStack.org/foundation/companies/>.

推广上, Redhat 与 Mirantis 无疑是两大领导厂商。Redhat 凭借自身在开源社区多年的沉淀, 携 RHEL 系列 Linux 系统及各种开源软件而拥有众多用户群体, 并且在 OpenStack 社区的代码贡献和部署推广上也一直处于领导地位; 而 Mirantis 虽然属于初创公司, 但是其凭借“Pure Play OpenStack”的理念, 完全从用户角度出发, 推出了一系列适合各种用户场景且简单易用的部署架构, 可以说 Mirantis 是最贴近用户实际、最全身心投入 OpenStack 的公司, 因此, 包括像大众汽车、爱立信等跨国大型企业均选用 Mirantis 方案进行云计算建设, 同时 Mirantis 也拥有极多的个人和中小企业用户。除了 Redhat 和 Mirantis, 如 HPE、Rackspace、Juniper 等企业也有自己的 OpenStack 高可用生产环境部署方案, 由于 OpenStack 组件功能的丰富多样, 每个企业的高可用部署方案可能不尽相同, 在后面的几节中, 我们将会概要介绍一些在社区里面比较具有代表性的企业 OpenStack 高可用部署方案。

2.5.1 Juniper Networks OpenStack 高可用部署方案

Juniper Networks 的 OpenStack 高可用方案由 OpenStack 和 Contrail 组件集合而成, Juniper Networks 的 SDN (Software Defined Network) 控制器包括 OpenStack 和 Contrail 两个部分, 因此 Juniper 的控制节点高可用设计由两个部分构成: OpenStack 的 HA 和 Contrail 的 HA。Juniper 的 OpenStack 高可用集群顶层架构如图 2-33 所示^①。

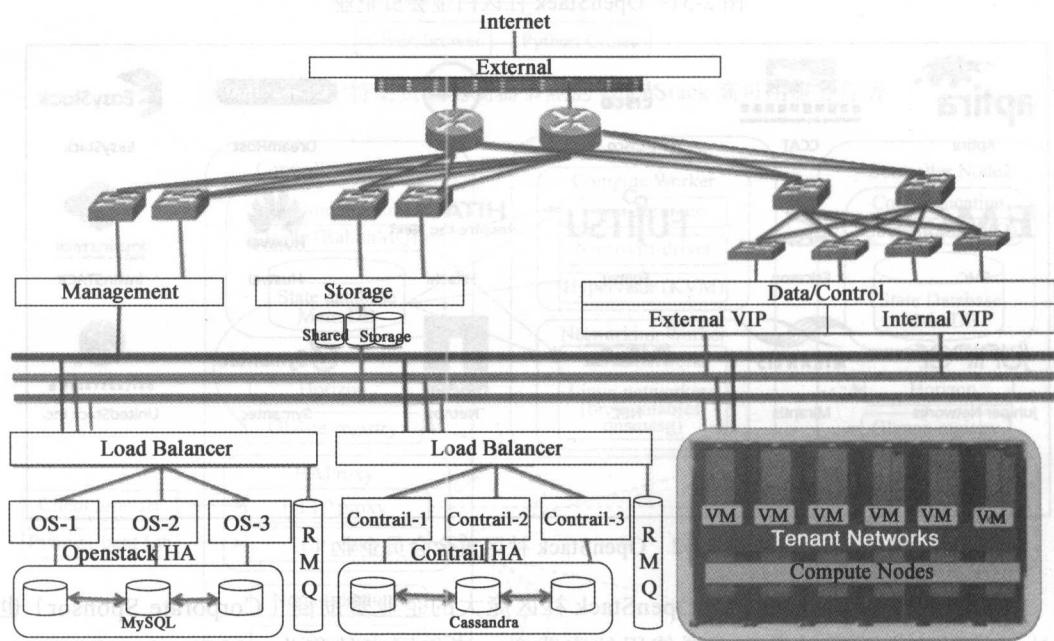


图 2-33 Juniper OpenStack 高可用集群部署架构

① Juniper Networks. Juniper OpenStack High Availability [EB/OL]. http://www.juniper.net/techpubs/en_US/contrail2.0/topics/task/configuration/juniper-high-availability-vnc.html.

Juniper OpenStack 高可用集群服务运行在 Active/Active 模式，并且可以实现基础架构和编排服务的 Scale-out 扩展，同时在 Juniper 架构的控制和编排层中引入新服务也非常方便，Juniper 的 OpenStack 高可用集群声称实现的目标包括：

- ❑ 5 个 9 (99.999%) 的高可用性；
- ❑ 任何时候都不会中断的云操作；
- ❑ 提供基于 VIP 访问的 API 和 UI；
- ❑ 跨集群的负载均衡网络操作；
- ❑ 弹性管理和部署；
- ❑ 故障检测和恢复。

在 Juniper 所实现的 OpenStack 高可用架构中，HAProxy 仍然是实现服务高可用的主要软件，OpenStack 部分的高可用和 Contrail 部分的高可用均是通过 HAProxy 的负载均衡实现，为了保证 VIP 的高可用，Juniper 引入了 Keepalived 机制，通过 VRRP 协议来实现 VIP 的高可用，图 2-34 是 Juniper 架构中 Opensatck 服务的高可用实现机制。

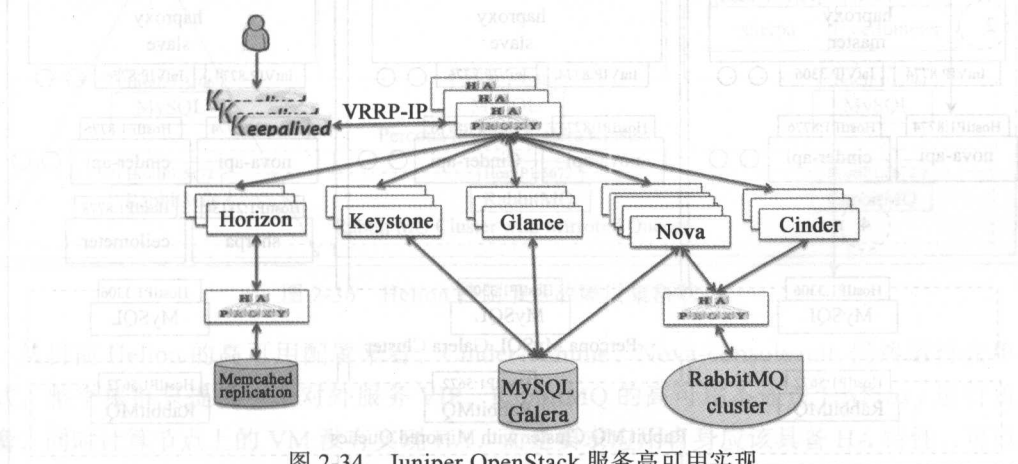


图 2-34 Juniper OpenStack 服务高可用实现

Juniper 的 OpenStack 高可用集群也存在一些限制，如只支持一个故障点发生，在 Failover 过程中，REST API 调用会失败，用户必须重新发起调用请求，在故障出现的时候并不能实现目标定义下的 100% 不丢包，同时 Juniper 高可用方案仅在 HAProxy 下测试通过，在使用其他第三方负载均衡软件的情况下高可用性没有经过测试。

2.5.2 HPE OpenStack 高可用部署方案

HP 在企业拆分之前便投入大量资源进行 OpenStack 的开发和部署，并推出了基于 OpenStack 的 Helion 公有云服务，拆分后的 HPE 仍然致力于 OpenStack 社区的发展和部署实践，目前的 Helion 便是由 HPE 进行运营维护。目前 HPE 的 Helion 高可用集群架构与 Redhat 的三节点高可用架构类似，高可用部分的软件栈也主要由 HAProxy 和 Keepalived 构

成，客户端对 OpenStack 服务的访问首先抵达处于 Active 状态的控制节点，然后被 HAProxy 捕获并终止，之后 HAProxy 再将访问请求转发到对应的 OpenStack 服务后端进行处理，而 Helion 外部 VIP 的高可用主要通过基于三节点的 Keepalived 集群实现。图 2-35 是 HPE 的 Helion 高可用集群架构图^①。在图 2-35 中，用户客户端对 OpenStack 服务的访问请求被发送到服务 VIP 及其端口，如访问 Nova 的服务，则请求被发送到 192.0.26:8447，HAProxy 随时在监听 VIP 及其端口上的请求访问，一旦发现有访问请求，HAProxy 便选择一个恰当的后端控制节点进行处理。

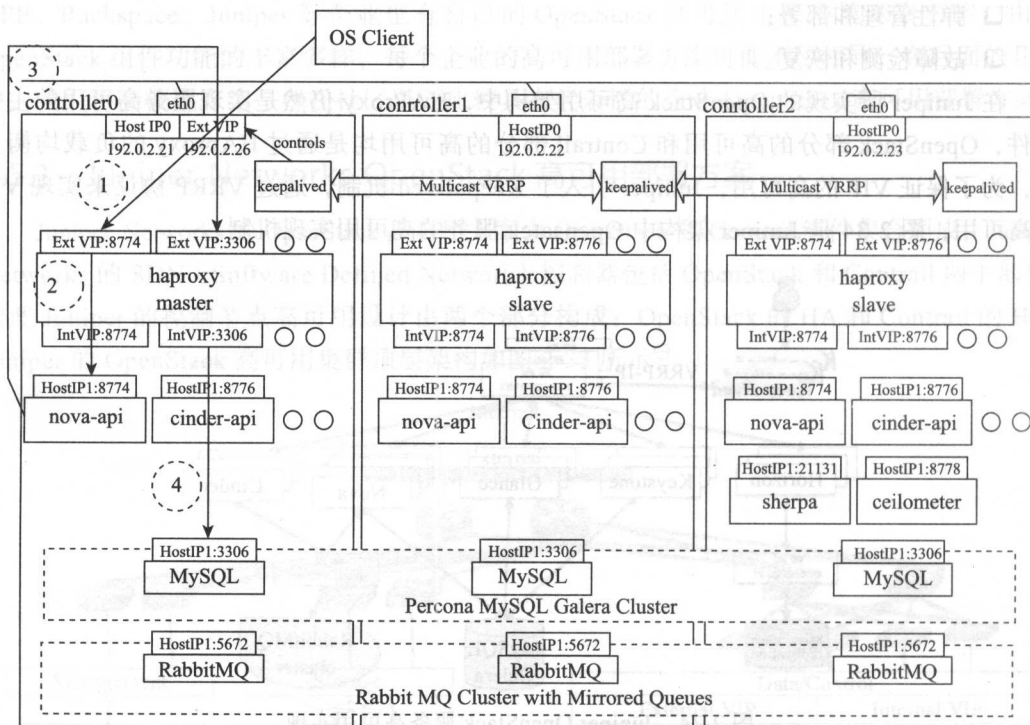


图 2-35 Helion OpenStack 集群高可用架构

在 HPE 的 Helion 中，OpenStack 的大多数 API 和 MySQL 均通过 HAProxy 实现了高可用，虽然 RabbitMQ 也实现了高可用，但是 RabbitMQ 的高可用并不是通过 HAProxy 实现，Helion 认为 RabbitMQ 的高可用应该通过客户端配置来实现，即在客户端配置文件中写入全部 RabbitMQ 服务器主机名称，由客户端自行选择可用 RabbitMQ 服务器。如果 Helion 的某个控制节点发生故障，Keepalived 将会迅速将 VIP 迁移到其他节点，并且接受客户端访问请求，如图 2-36 所示，而整个过程对于客户端而言是透明的，即客户端对 OpenStack 的访问方式没有任何变化。如果之前的故障节点恢复并加入集群，则 Keepalived 会将该节

① Hewlett Packard Enterprise. HPE Helion OpenStack 1.1 High Availability [EB/OL]. <https://docs.hpcloud.com/#commercial/GA1/1.1commercial.high-availability.html>.

点置为 Standby/Slave 状态，即当前活动并接受访问请求的控制节点并不会发生变化，如图 2-37 所示。

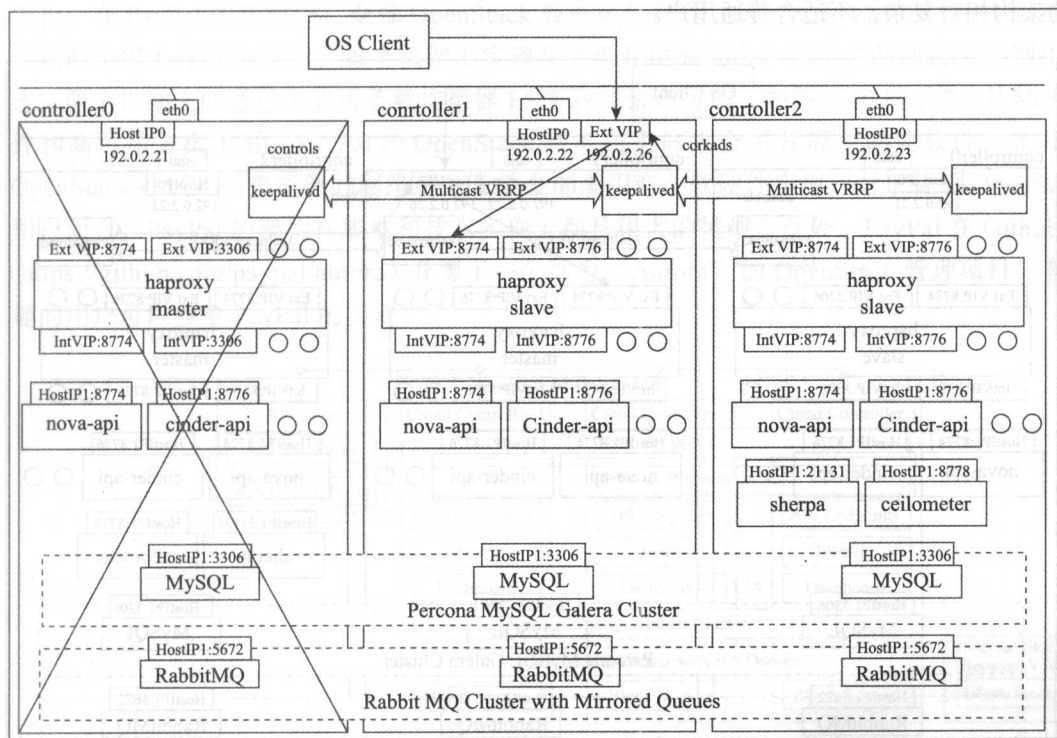


图 2-36 Helion 控制节点故障后集群状态

从目前 Helion 的高可用配置来看，Cinder-volume、Nova-consoleauth 仍然运行在单点模式，整个集群只提供一个对外服务 VIP，RabbitMQ 的高可用不通过 HAProxy 进行负载均衡，同时计算节点上的 VM 没有实现 HA，而是要求应用自身应该具备 HA 特性，可以说 Helion 实现了大多数 OpenStack 服务的高可用性，但是相比 Redhat 和 Mirantis 的方案，还不能算是比较理想和彻底的高可用方案。

2.5.3 TCP Cloud OpenStack 高可用部署方案

TCPCloud 的 OpenStack 高可用方案主要针对企业私有云设计，与 Juniper Networks 的架构类似，TCPCloud 的网络控制器 SDN 也由两部分组成，即 OpenStack 和 OpenContrail，因此在高可用架构实现上，也分为 OpenStack 的 HA 和 OpenContrail 的 HA，图 2-38 为 TCPCloud 的 OpenStack 高可用架构。TCPCloud 最新的 OpenStack 发行版本为 Mk.20，主要基于 OpenStack Kilo 版本和 OpenContrail R2.2 版本以及其他社区开源软件所构建。Mk.20 资源高可用的实现主要是基于 HAProxy 和 Keepalived，而不再基于之前的 Pacemaker/Corosync 技术栈，Keystone 的 Token 使用的是 Fernet Tokens 而不再是 UUID Tokens。

Neutron 的网络后端为 OpenContrail R2.2, Cinder 后端主要是 IBM 的中端存储 Stowize 系列, 并采用 SlatStack 进行配置管理, Foreman 进行生命周期管理^①, 总体而言, TCPCloud 的架构相对复杂, 不适合普通用户。

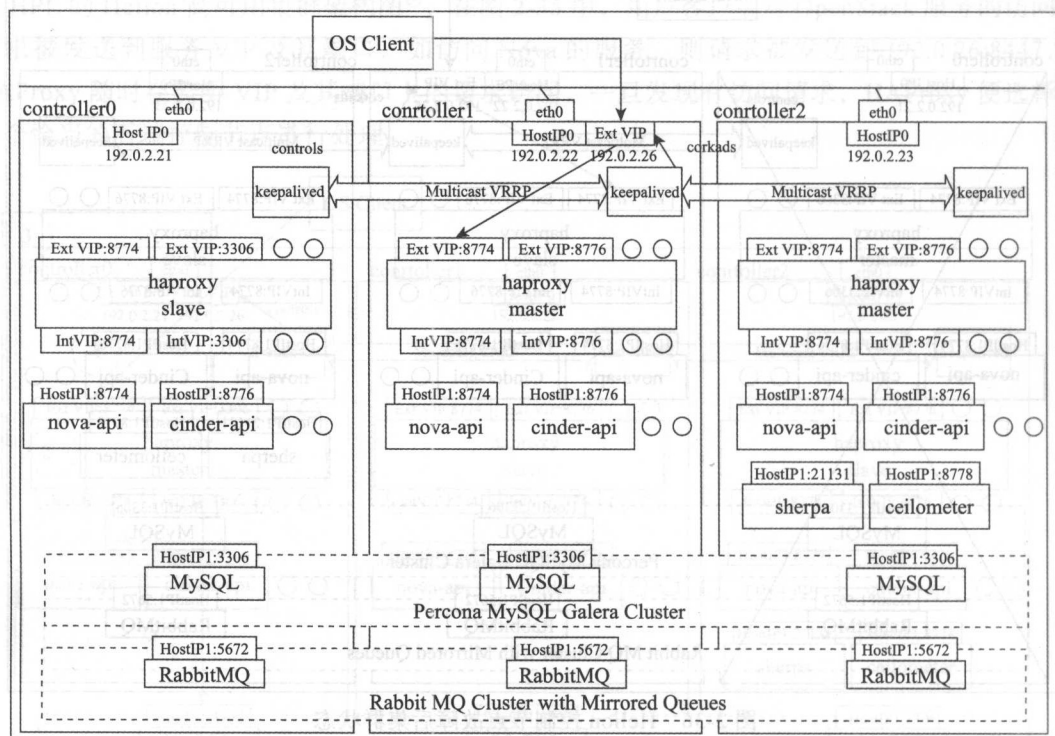


图 2-37 Helion 故障节点恢复后集群状态

2.5.4 Paypal OpenStack 高可用部署方案

PayPal 面向全球众多用户提供国际贸易支付工具, 其官方数据表明 PayPal 的活跃用户为 1.73 亿, 可以在 203 个国家提供 26 种货币交易, 支撑 PayPal 如此庞大交易量的后端系统中便有 OpenStack 的功劳, 而 PayPal 弃用 VMware 转向 OpenStack, 曾造就了最大的 OpenStack 金融云案例。作为应用到金融支付领域的 OpenStack 私有云, PayPal 的 OpenStack 高可用架构 (图 2-39) 不仅使用到了 OpenStack 社区高可用性部署的通用最佳实践, 还结合了传统数据中心高可用性建设的经验, 因此, PayPal 的 OpenStack 高可用集群架构不仅仅是开源软件的堆栈, 而且还包括很多商业设备和软件。图 2-39 中, PayPal 的高可用架构的网络负载部分采用了 F5 公司的负载均衡服务器, 而网络 SDN 部分则采用的是 Nicira 基于 Openflow 和 OpenvSwitch 开发的 NVP (网络虚拟平台) 技术。此外, PayPal 还

^① TCP Cloud. Tcp cloud Release Mk. 20 [EB/OL]. <http://www.opentcpcloud.org/en/release-notes/release-mk20/>.

将集群从物理机架上分为 Infrastructure Rack 和 Compute Rack, Infrastructure Rack 上的服务器只提供云管理服务, 而 Compute Rack 可以不断扩展直到 NVP 网关耗尽和 IP 地址用完为止。在 PayPal 的架构中, 全部 OpenStack 服务运行在基于 KVM 的 VM 中, 每个服务至少在两个以上 VM 上运行, 每个机架上实现交换机和电源冗余, 机架之间通过三层网络互连, 两个机架之间通过企业级负载均衡器 F5 实现冗余高可用, 如图 2-40 所示^①。从部署组件和高可用方案上看, PayPal 的 OpenStack 高可用集群集合了开源与商业软件, 尤其是 OpenStack 社区不是很成熟的网络部分几乎全部采用商业软硬件实现, 对于普通 OpenStack 用户而言, PayPal 的架构在成本和技术实现上都是很大的挑战。此外, PayPal 在 Github 上 (<https://github.com/paypal/aurora>) 开源了一个称为“Aurora”的 OpenStack 管理项目, 有兴趣的用户可以参考 PayPal 的实现。

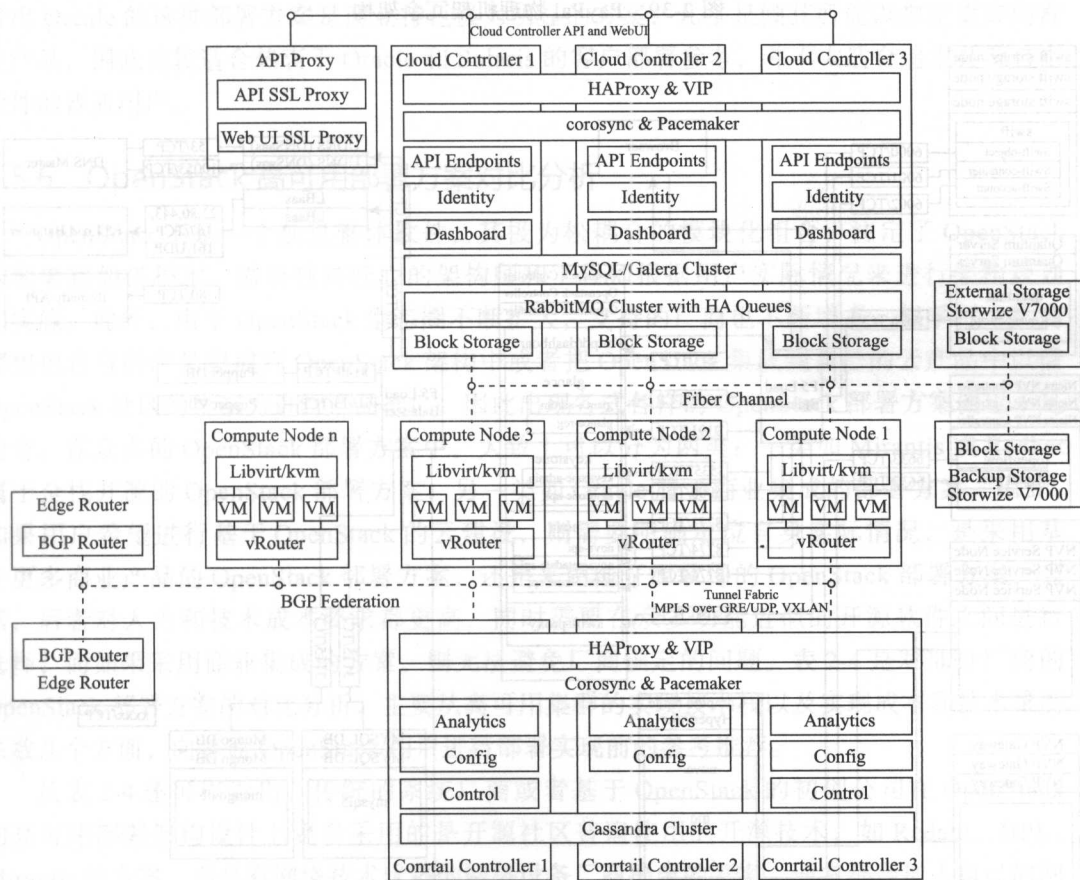


图 2-38 TCPCloud OpenStack 高可用集群架构

① PayPal. OpenStack Summit HK 2013 High Availv 5-1. pptx [EB/OL]. OpenStack. org. 2013. <https://www.OpenStack.org/assets/presentation-media/OpenStackSummitHK2013HighAvailv5-1.pptx>.

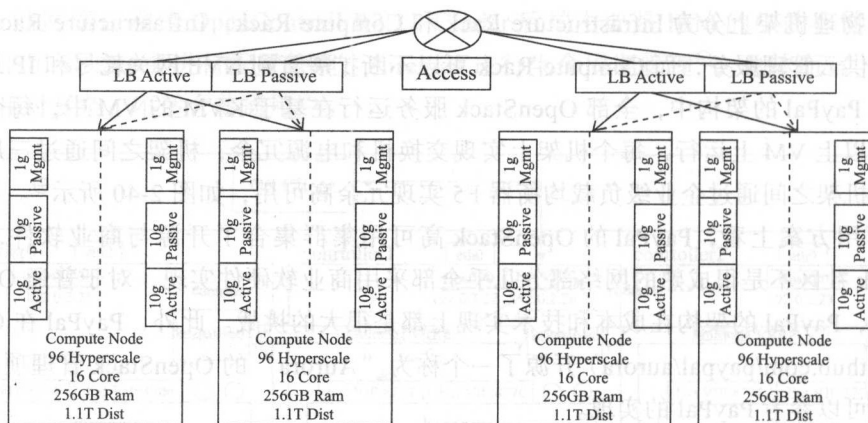


图 2-39 PayPal 物理机架冗余架构

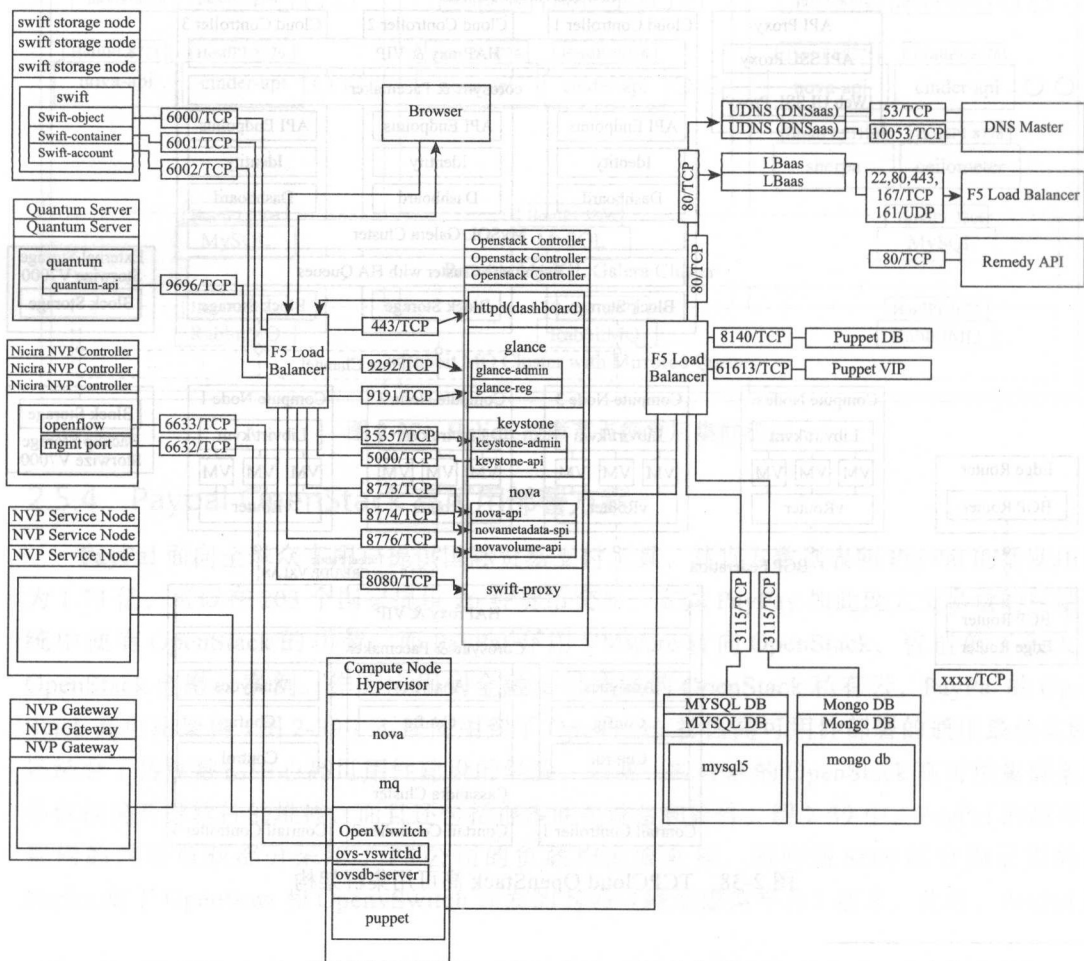


图 2-40 PayPal OpenStack 高可用集群部署架构

2.5.5 Oracle OpenStack 高可用部署方案

Oracle 收购 Sun 之后，作为 Solaris 的维护者，基于 Solaris 系统和 Solaris 集群管理软件也提出了自己的 OpenStack 高可用部署方案（见图 2-41）。在这类高可用方案中，除了系统和高可用集群软件使用的是 Oracle 自己的产品外，Oracle 的集群共享存储也通过 Oracle ZFS Storage Appliance（ZFS SA）来提供，而 Cinder 的 Volume 块存储则通过 Oracle ZFS Storage Appliance iSCSI Cinder driver 驱动来使用 ZFS SA 共享存储^①。Oracle 将在两台主机上部署的 MySQL 服务划为一个特定域，RabbitMQ 服务又划为一个特定域，全部的 OpenStack 服务划为独立特定域，EVS（ElasticVirtual Switch）控制器和 Neutron-agent 位于全局集群域（Global cluster），并且 EVS 位于一个 Failover 域中可以实现 Failover，而 OpenStack 服务的高可用性通过 Solaris 的 SMF（Service Management Facility）实现。可以看出 Oracle 的这种部署方案是商业特色极为明显的方案，几乎是倾其所能以绑定更多的商业产品，因此比较适合热衷于 Oracle 和 Solaris 的客户部署参考，并不太适合追求开源自由软件的普通用户。

2.5.6 OpenStack 高可用部署方案对比分析

OpenStack 不是一个独立整体软件，其极为松耦合的模块化组合，注定了 OpenStack 的部署犹如搭积木，需要独具匠心的架构师和工程师根据用户实际情况来进行架构规划和实施。此外，由于 OpenStack 生态圈不断扩大，支持的厂商也不断增多，似乎每个厂商都想把自身的产品集成到 OpenStack 架构中或者把 OpenStack 集成到自己的云产品中以借 OpenStack 社区的影响力进行产品推广，因此出现各式各样的 OpenStack 部署方案便也不足为奇。在众多的 OpenStack 部署方案中，大致上可以分为两类：一类如 Mirantis 或 Redhat 属于全栈开源的 OpenStack 部署方案；另一类如 Oracle 属于商业集成的部署方案。因此，如果用户希望进行基于 OpenStack 的云建设，则需要明确定位自身实际情况，是采用基于更多商业产品的 OpenStack 部署方案，还是采用纯开源软件的 OpenStack 部署方案，当然，后者对人力和技术成本要求都更高，同时需要在众多功能近似的开源软件之间进行选择，而如果采用商业集成的方案，则无法避免厂商锁定的问题。表 2-4 是对部分厂商的 OpenStack 部署方案的对比分析，主要从高可用集群的实现技术栈以及实现成本和技术难度系数几个方面，向普通 OpenStack 用户进行部署实现前的参考推荐。

从表 2-4 还可以看出，传统的系统厂商或者基于 OpenStack 的初创公司在 OpenStack 的高可用部署架构设计上更多采用的是开源社区普遍使用的开源技术，如 Redhat、HPE、Mirantis 的方案，而具有网络技术优势的网络设备厂商提出的方案，通常都会加入自己的网络组件以增强集群网络的高可用性，如 TCP Cloud 和 Juniper Networks 的方案，其他商业氛

① Oracle. Oracle OpenStack for Oracle Linux High Availability Guide [J/OL]. Oracle. 2014. <http://www.oracle.com/technetwork/server-storage/OpenStack/linux/documentation/ha-guide-oracle-OpenStack-2296039.pdf>

围比较重的公司如 Oracle 和 PayPal 则更多的是集成自己的商业产品或者采购的商业产品来实现 OpenStack 集群的高可用。

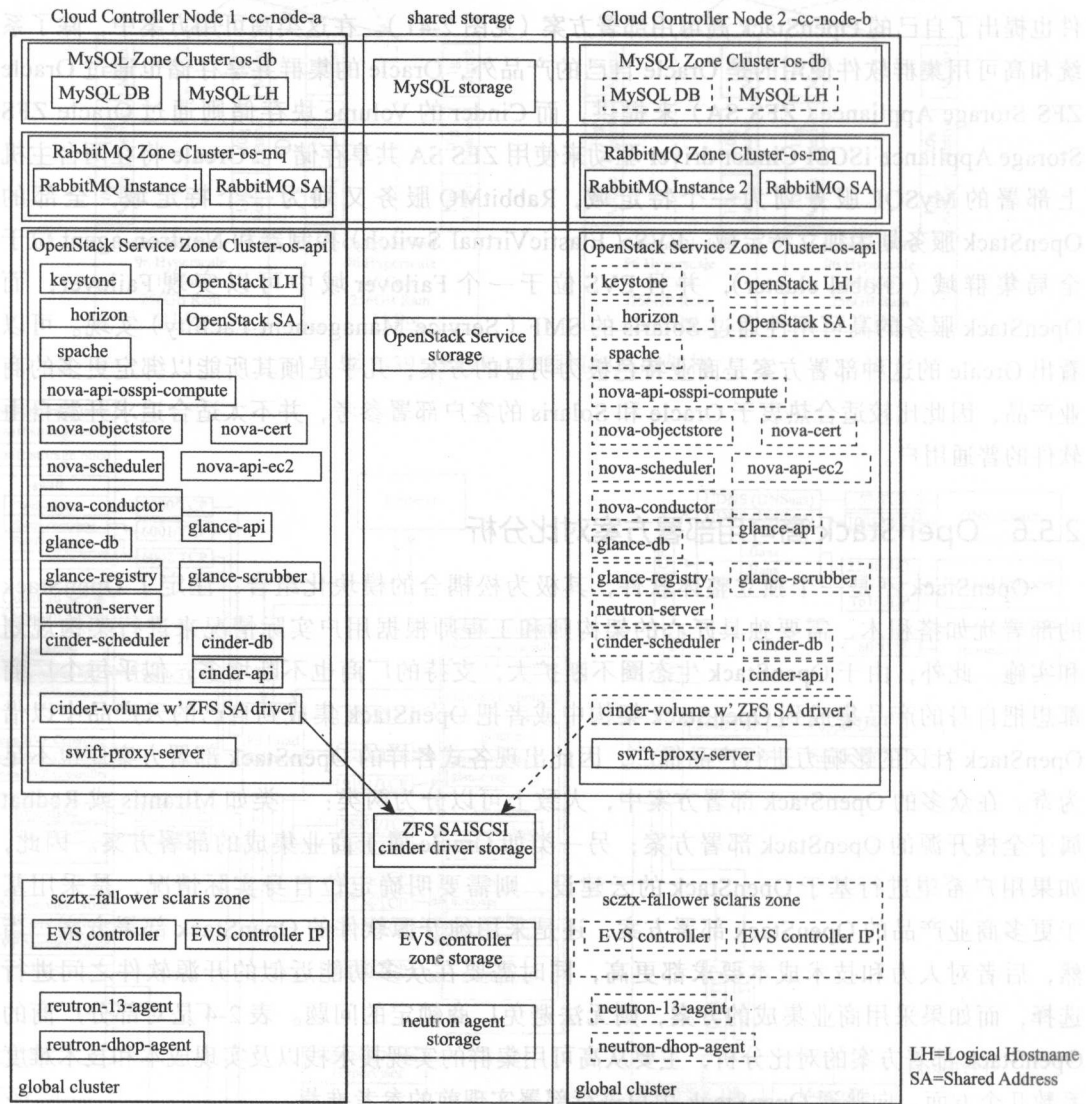


图 2-41 Oracle 基于 Solaris 的 OpenStack 高可用部署方案

表 2-4 各厂商的 OpenStack 高可用部署方案的对比分析

厂 商	集群高可用技术	开源实现 / 商业集成	技术考虑	成本考虑	综合考虑
Redhat	Pacemaker/Corosync/Galera/HAProxy	开源实现	适中	较低	推荐
Mirantis	Pacemaker/Corosync/Galera/HAProxy	开源实现	简单	较低	推荐
Juniper	HAProxy/Keepalived/Contrail/Galera	开源实现	较难	适中	可考虑

(续)

厂 商	集群高可用技术	开源实现 / 商业集成	技术考 虑	成本考虑	综合考虑
HPE	HAProxy/Keepalived/ Galera	开源实现	适中	较低	可考虑
TCPCloud	Pacemaker/Corosync/HAProxy/Galera/Contrail	开源实现	较难	适中	可考虑
PayPal	F5/Nicira NVP/ Galera	商业集成	很难	较高	不推荐
Oracle	Oracle Solaris Cluster	商业集成	很难	较高	不推荐

2.6 本章小结

本章主要介绍了 OpenStack 高可用集群部署中的节点功能组件和服务功能组件，并着重对 OpenStack 集群中的计算节点、控制节点、存储节点、网络节点和负载均衡器进行了介绍，同时对 Nova、Cinder、Glance、Keystone、Neutron 等服务组件进行了介绍。此外，本章还重点阐述了 OpenStack 两大领导厂商 Redhat 与 Mirantis 的高可用部署方案，同时也概念性地介绍了 HPE、Juniper、TCPCloud、PayPal、Oracle 等厂商的 OpenStack 高可用部署方案，最后，本章对各个厂商的 OpenStack 高可用部署方案进行了一般性的对比，以供准备部署实施 OpenStack 高可用集群的用户选择参考。

原 理 篇

- 第3章 集群资源管理系统
- 第4章 集群负载均衡系统
- 第5章 集群消息队列系统
- 第6章 集群缓存系统
- 第7章 集群数据库系统
- 第8章 OpenStack 计算服务
- 第9章 OpenStack 网络服务
- 第10章 OpenStack 存储服务

集群资源管理系统

云计算与集群系统密不可分，作为分布式计算和集群计算的集大成者，云计算的基础设施必须通过集群进行管理控制，而作为拥有大量资源与节点的集群，必须具备一个强大的集群资源管理器（Cluster System Manager, CSM）来调度和管理集群资源。对于任何集群而言，集群资源管理器是整个集群能够正常运转的大脑和灵魂，任何集群资源管理器的缺失和故障都会导致集群陷入瘫痪混乱的状态。OpenStack 的众多组件服务既可以集成到单个节点上运行，也可以在集群中分布式运行。但是，要实现承载业务系统的高可用集群，OpenStack 服务必须部署到高可用集群上，并在实现 OpenStack 服务无单点故障的同时，实现故障的自动转移和自我愈合，而这些功能是 OpenStack 的多数服务本身所不具备的。因此，在生产环境中部署 OpenStack 高可用集群时，必须引入第三方集群资源管理软件，专门负责 OpenStack 集群资源的高可用监控调度与管理。

集群资源管理软件种类繁多，并有商业软件与开源软件之分。在传统业务系统的高可用架构中，商业集群管理软件的使用非常普遍，如 IBM 的集群系统管理器、PowerHA SystemMirror（也称为 HACMP）以及针对 DB2 的 PureScale 数据库集群软件；再如 Oracle 的 Solaris Cluster 系列集群管理软件，以及 Oracle 数据库的 ASM 和 RAC 集群管理软件等商业高可用集群软件都在市场上占有很大的比例。此外，随着开源社区的发展和开源生态系统的扩大，很多商业集群软件也正在朝着开源的方向发展，如 IBM 开源的 xCAT 集群软件。而在 Linux 开源领域，Pacemaker/Corosync、HAProxy/Keepalived 等组合集群资源管理软件也有着极为广泛的应用。

在第 2 章中，我们曾列举了包括 Redhat 和 Mirantis 等 OpenStack 领导厂商的 OpenStack 高可用集群部署方案，各个厂商的 OpenStack 高可用部署方案表明，Pacemaker/Corosync 和 HAProxy/Keepalived 已经成为 OpenStack 高可用集群部署的集群资源管理器和负载均衡

器的标准，而 Pacemaker 作为 Linux 集群资源高可用的管理软件，已被 OpenStack 官方社区指定为 OpenStack 集群高可用部署的集群资源管理器。本章将以 Pacemaker 为基础，重点介绍 OpenStack 高可用集群部署中的集群管理系统，本章内容也是部署 OpenStack 集群的基础，其资源管理部分的内容将会贯穿整个 OpenStack 高可用集群的部署与运维过程中。

3.1 Pacemaker 概述

Pacemaker 是 Linux 环境中使用最为广泛的开源集群资源管理器，Pacemaker 利用集群基础架构（Corosync 或者 Heartbeat）提供的消息和集群成员管理功能，实现节点和资源级别的故障检测和资源恢复，从而最大程度保证集群服务的高可用。从逻辑功能而言，Pacemaker 在集群管理员所定义的资源规则驱动下，负责集群中软件服务的全生命周期管理，这种管理甚至包括整个软件系统以及软件系统彼此之间的交互。Pacemaker 在实际应用中可以管理任何规模的集群，由于其具备强大的资源依赖模型，这使得集群管理员能够精确描述和表达集群资源之间的关系（包括资源的顺序和位置等关系）。同时，对于任何形式的软件资源，通过为其自定义资源启动与管理脚本（资源代理），几乎都能作为资源对象而被 Pacemaker 管理。此外，需要指出的是，Pacemaker 仅是资源管理器，并不提供集群心跳信息，由于任何高可用集群都必须具备心跳监测机制，因而很多初学者总会误以为 Pacemaker 本身具有心跳检测功能，而事实上 Pacemaker 的心跳机制主要基于 Corosync 或 Heartbeat 来实现。

从起源上来看，Pacemaker 是为 Heartbeat 项目而开发的 CRM 项目的延续，CRM 最早出现于 2003 年，是专门为 Heartbeat 项目而开发的集群资源管理器，而在 2005 年，随着 Heartbeat2.0 版本的发行才正式推出第一版本的 CRM，即 Pacemaker 的前身。在 2007 年末，CRM 正式从 Heartbeat2.1.3 版本中独立，之后于 2008 年 Pacemaker0.6 稳定版本正式发行，随后的 2010 年 3 月 CRM 项目被终止，作为 CRM 项目的延续，Pacemaker 被继续开发维护，如今 Pacemaker 已成为开源集群资源管理器的事实标准而被广泛使用。此外，Heartbeat 到了 3.0 版本后已经被拆分为几个子项目了，这其中便包括 Pacemaker、Heartbeat3.0、Cluster Glue 和 Resource Agent，这几个子项目之间的关系如图 3-1 所示。

(1) Heartbeat

Heartbeat 项目最初的消息通信层被独立为新的 Heartbeat 项目，新的 Heartbeat 只负责维护集群各节点的信息以及它们之间的心跳通信，通常将 Pacemaker 与 Heartbeat 或者 Corosync 共同组成集群管理软件，Pacemaker 利用 Heartbeat 或者 Corosync 提供的节点及节点之间的心跳信息来判断节点状态。

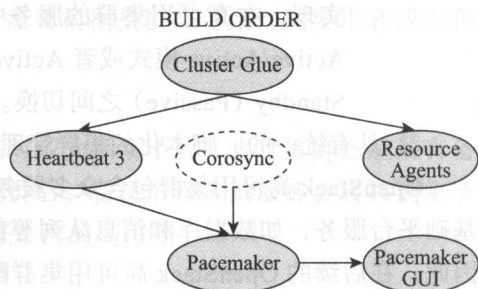


图 3-1 Pacemaker 及相关组件功能拓朴

(2) Cluster Glue

Cluster Glue 相当于一个中间层,它用来将 Heartbeat 和 Pacemaker 关联起来,主要包含两个部分,即本地资源管理器(Local Resource Manager, LRM)和 Fencing 设备(Shoot The Other Node In The Head, STONITH)。

(3) Resource Agent

资源代理(Resource Agent, RA)是用来控制服务启停、监控服务状态的脚本集合,这些脚本会被位于本节点上的 LRM 调用从而实现各种资源的启动、停止、监控等操作。

(4) Pacemaker

Pacemaker 是整个高可用集群的控制中心,用来管理整个集群的资源状态行为,客户端通过 Pacemaker 来配置、管理、监控整个集群的运行状态。

Pacemaker 是一个功能非常强大并支持众多操作系统的开源集群资源管理器, Pacemaker 支持主流的 Linux 系统,如 Redhat 的 RHEL 系列、Fedora 系列、OpenSUSE 系列、Debian 系列、Ubuntu 系列和 CentOS 系列,这些操作系统上都可以运行 Pacemaker 并将其作为集群资源管理器。Pacemaker 的主要功能包括以下几方面:

- ❑ 监测并恢复节点和服务级别的故障。
- ❑ 存储无关,并不需要共享存储。
- ❑ 资源无关,任何能用脚本控制的资源都可以作为集群服务。
- ❑ 支持节点 STONITH 功能以保证集群数据的完整性和防止集群脑裂。
- ❑ 支持大型或者小型集群。
- ❑ 支持 Quorum 机制和资源驱动类型的集群。
- ❑ 支持几乎是任何类型的冗余配置。
- ❑ 自动同步各个节点的配置文件。
- ❑ 可以设定集群范围内的 Ordering、Colocation and Anti-colocation 等约束。
- ❑ 高级服务类型支持,例如:
 - Clone 功能,即那些要在多个节点运行的服务可以通过 Clone 功能实现,Clone 功能将会在多个节点上启动相同的服务;
 - Multi-State 功能,即那些需要运行在多状态下的服务可以通过 Multi-State 来实现,在高可用集群的服务中,有很多服务会运行在不同的高可用模式下,如 Active/Active 模式或者 Active/Passive 模式等,并且这些服务可能会在 Active 与 Standby (Passive) 之间切换。
- ❑ 具有统一的、脚本化的集群管理工具。

OpenStack 高可用集群包含众多服务,除了 OpenStack 自身的诸多服务以外,还有很多基础平台服务,如数据库和消息队列等都需要使用 Pacemaker 进行资源高可用的管理控制。因此,在后续的 OpenStack 高可用集群配置中,几乎会使用到 Pacemaker 提供的上述功能。比如 Keystone 与 Nova 等 API 服务在集群中的启动顺序将由 Pacemaker 的 Ordering 约束来

控制,而某些服务需要同时运行在某个节点上,这时将会用到 Pacemaker 的 Colocation 约束。另外,由于 MariaDB 和 RabbitMQ 是多状态服务,因此需要 Pacemaker 的 Multi-State 高级功能的支持。总之, Pacemaker 提供了丰富的集群功能来充分满足用户对集群节点和服务所进行的各种自定义高可用设置,从而最终实现集群服务的高可用性。

3.2 Pacemaker 集群分类

Pacemaker 对用户的环境没有特定的要求,这使得它支持任何类型的高可用节点冗余配置,包括 Active/Active、Active/Passive、N+1、N+M、N-to-1 and N-to-N 模式的高可用集群,用户可以根据自身对业务的高可用级别要求和成本预算,通过 Pacemaker 部署适合自己的高可用集群。

(1) Active/Active 模式

在这种模式下,故障节点上的访问请求或自动转到另外一个正常运行节点上,或通过负载均衡器在剩余的正常运行的节点上进行负载均衡。这种模式下集群中的节点通常部署了相同的软件并具有相同的参数配置,同时各服务在这些节点上并行运行。

(2) Active/Passive 模式

在这种模式下,每个节点上都部署有相同的服务实例,但是正常情况下只有一个节点上的服务实例处于激活状态,只有当前活动节点发生故障后,另外的处于 Standby 状态的节点上的服务才会被激活,这种模式通常意味着需要部署额外的且正常情况下不承载负载的硬件。

(3) N+1 模式

所谓的 N+1 就是多准备一个额外的备机节点,当集群中某一节点故障后该备机节点会被激活从而接管故障节点的服务。在不同节点安装和配置有不同软件的集群中,即集群中运行有多个服务的情况下,该备机节点应该具备接管任何故障服务的能力,而如果整个集群只运行同一个服务,则 N+1 模式便退变为 Active/Passive 模式。

(4) N+M 模式

在单个集群运行多种服务的情况下, N+1 模式下仅有的一个故障接管节点可能无法提供充分的冗余,因此,集群需要提供 M (M>1) 个备机节点以保证集群在多个服务同时发生故障的情况下仍然具备高可用性, M 的具体数目需要根据集群高可用性的要求和成本预算来权衡。

(5) N-to-1 模式

在 N-to-1 模式中,允许接管服务的备机节点临时成为活动节点(此时集群已经没有备机节点),但是,当故障主节点恢复并重新加入到集群后,备机节点上的服务会转移到主节点上运行,同时该备机节点恢复 Standby 状态以保证集群的高可用。

(6) N-to-N 模式

N-to-N 是 Active/Active 模式和 N+M 模式的结合, N-to-N 集群将故障节点的服务和访

问请求分散到集群其余的正常节点中，在 N-to-N 集群中并不需要有 Standby 节点的存在，但是需要所有 Active 节点均有额外的剩余可用资源。

在实际的高可用集群部署中，两节点主备高可用模式（Active/Passive）是一种较为常见的部署模式，其架构如图 3-2 所示。在 Active/Passive 模式下，只有主节点运行服务，备节点处于 Standby 模式，当主节点发生故障时，备节点将迅速接管故障服务并对外提供访问。在 Linux 环境中，使用 Pacemaker 和 DRBD 的双节点主备方案作为一种高效、经济的开源解决方案在很多企业高可用环境中被采用，其高可用模式如图 3-3 所示，当 Active 节点故障后，共享存储锁将被释放，与此同时 Standby 节点将挂载共享存储，之后与故障节点相同的服务将在 Passive 节点重新启动并对外提供服务。

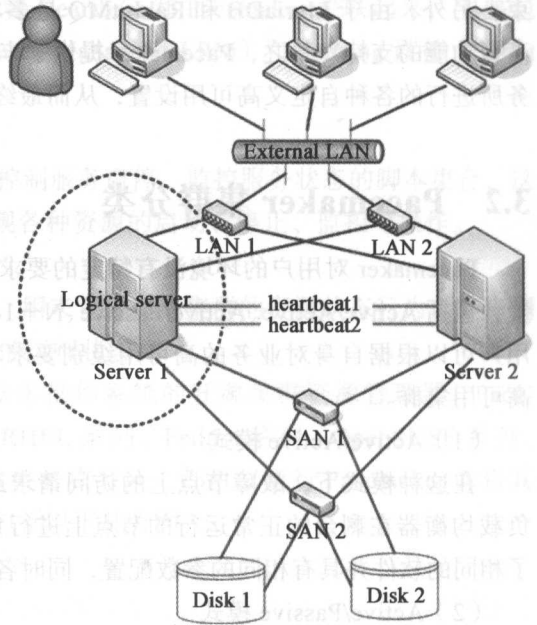


图 3-2 常见两节点高可用集群架构

在 Active/Passive 模式中，如果需要配置多个独立 Cluster，而每个 Cluster 又都配置一个 Standby 节点，则势必对物理资源造成极大浪费，因为 Standby 节点在多数时间均处于空闲状态。而在 Pacemaker 集群中，实现了多集群共享 Standby 节点，即多个 Cluster 同时使用一个 Standby 节点，从而使得 Standby 节点发挥了最大利用价值并最终减少硬件资源浪费，共享 Standby 节点的 Pacemaker 主备高可用集群如图 3-4 所示。

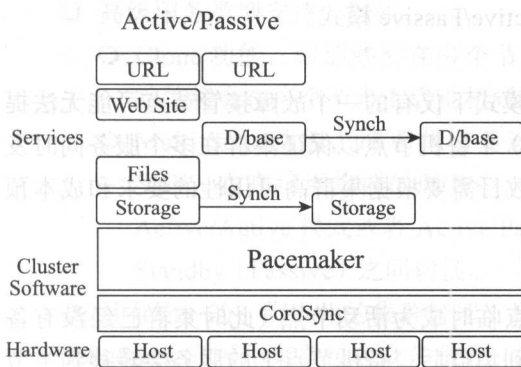


图 3-3 Active/Passive 模式高可用集群

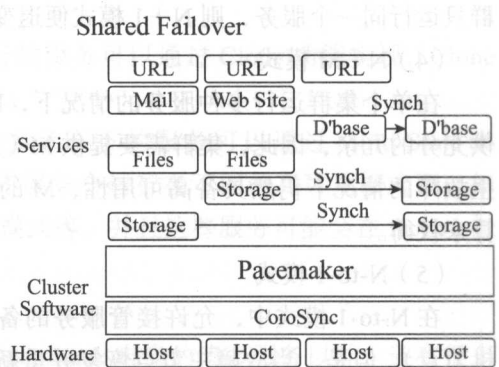


图 3-4 共享 Standby 节点的 Active/Passive 模式高可用集群

此外，如果使用分布式共享存储，则 Pacemaker 也支持 Active/Active 模式，即相同的

服务同时运行在集群中的多个节点上,并且每个节点都可以接管故障节点的服务(N-to-N模式),在这种模式下,Pacemaker在多个节点上同时运行服务副本从而实现对外服务请求的负载均衡,N-to-N模式如图3-5所示。

3.3 Pacemaker 集群架构

从高层次的集群抽象功能来看,Pacemaker的核心架构主要由集群不相关组件、集群资源管理组件和集群底层基础模块三个部分组成。

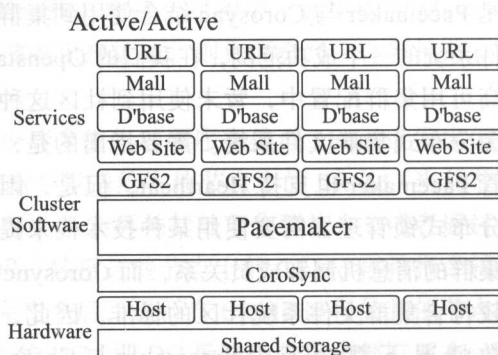


图 3-5 N-to-N 模式高可用集群

(1) 底层基础模块

底层的基础架构模块主要向集群提供可靠的消息通信、集群成员关系和 Quorum 等功能,底层基础模块主要包括像 Corosync、CMAN 和 Heartbeat 等项目组件。

(2) 集群无关组件

在 Pacemaker 架构中,这部分组件主要包括资源本身以及用于启动、关闭以及监控资源状态的脚本,同时还包括用于屏蔽和消除实现这些脚本所采用的不同标准之间差异的本地进程。虽然在运行多个实例时,资源彼此之间的交互就像一个分布式的集群系统,但是,这些实例服务之间仍然缺乏恰当的 HA 机制和独立于资源的集群治理能力,因此还需要后续集群组件的功能支持。

(3) 资源管理

Pacemaker 就像集群大脑,专门负责响应和处理与集群相关的事件,这些事件主要包括集群节点的加入、集群节点脱离,以及由资源故障、维护、计划的资源相关操作所引起的资源事件,同时还包括其他的一些管理员操作事件,如对配置文件的修改和服务重启等操作。在对所有这些事件的响应过程中,Pacemaker 会计算出当前集群应该实现的最佳理想状态,并规划出实现该理想状态后续需要进行的各种集群操作,这些操作可能包括了资源移动、节点停止,甚至包括使用远程电源管理模块来强制节点下线等。Pacemaker 的架构组成如图 3-6 所示。

当 Pacemaker 与 Corosync 集成时,Pacemaker 也支持常见的主流开源集群文件系统,而根据集群文件系统社区过去一直从事的标准化工作,社区使用了一种通用的分布式锁管理器来实现集群文件系统的并行读写访问,这种分布式锁控制器利用了 Corosync 所提供的集群消息和集群成员节点处理能力(节点是在线或离线的状态)来实现文件系统集群,同时使用 Pacemaker 来对服务进行隔

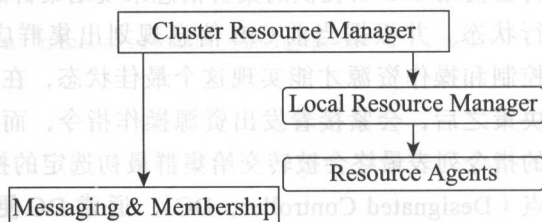


图 3-6 Pacemaker 架构组成

离 (Fencing) 操作, 在这种使用方式下, Pacemaker 的组件构成如图 3-7 所示。当然, 这只是 Pacemaker 与 Corosync 结合使用到集群文件系统的一个成功范例, 在我们的 Openstack 高可用集群配置中, 暂未使用到社区这种开源分布式集群文件系统。需要指出的是, 尽管 Pacemaker 也支持 Heartbeat, 但是, 因为分布式锁管理器需要使用某种技术栈来提供集群的消息机制和成员关系, 而 Corosync 比较符合集群文件系统社区的标准, 因此, 多数情况下都会采用 Pacemaker 与 Corosync 的组合而不是与 Heartbeat 的组合, 不过从技术角度而言, 在开源集群文件系统中使用 Heartbeat 也是可行的。

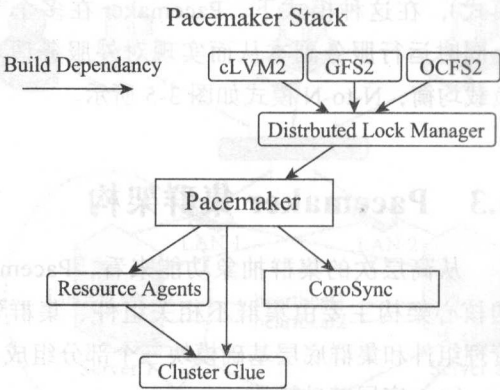


图 3-7 集成并行文件系统的 Pacemaker 组件构成

3.4 Pacemaker 内部组件

Pacemaker 作为一个独立的集群资源管理器项目, 其本身由多个内部组件构成, 这些内部组件彼此之间相互通信协作并最终实现了集群的资源管理, Pacemaker 项目由五个内部组件构成, 各个组件之间的关系如图 3-8 所示。

- ❑ CIB: 集群信息基础 (Cluster Information Base)。
- ❑ CRMD: 集群资源管理进程 (Cluster Resource Manager daemon)。
- ❑ LRMd: 本地资源管理进程 (Local Resource Manager daemon)。
- ❑ PEngine (PE): 策略引擎 (PolicyEngine)。
- ❑ STONITHd: 集群 Fencing 进程 (Shoot The Other Node In The Head daemon)。

CIB 主要负责集群最基本的信息配置与管理, Pacemaker 中的 CIB 主要使用 XML 的格式来显示集群的配置信息和集群所有资源的当前状态信息。CIB 所管理的配置信息会自动在集群节点之间进行同步, PE 将会使用 CIB 所提供的集群信息来规划集群的最佳运行状态。并根据当前 CIB 信息规划出集群应该如何控制和操作资源才能实现这个最佳状态, 在 PE 做出决策之后, 会紧接着发出资源操作指令, 而 PE 发出的指令列表最终会被转交给集群最初选定的控制器节点 (Designated Controller, DC), 通常 DC 便是运行 Master CRMD 的节点。在集群启动之初, Pacemaker 便

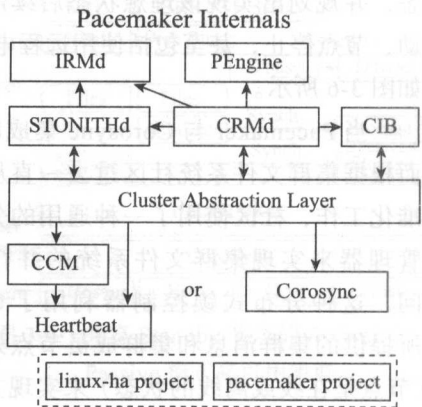


图 3-8 Pacemaker 内部组件

会选择某个节点上的 CRM 进程实例来作为集群 Master CRMd，然后集群中的 CRMd 便会集中处理 PE 根据集群 CIB 信息所决策出的全部指令集。在这个过程中，如果作为 Master 的 CRM 进程出现故障或拥有 Master CRM 进程的节点出现故障，则集群会马上在其他节点上重新选择一个新的 Master CRM 进程。

在 PE 的决策指令处理过程中，DC 会按照指令请求的先后顺序来处理 PEngine 发出的指令列表，简单来说，DC 处理指令的过程就是把指令发送给本地节点上的 LRMd（当前节点上的 CRMd 已经作为 Master 在集中控制整个集群，不会再并行处理集群指令）或者通过集群消息层将指令发送给其他节点上的 CRMd 进程，然后这些节点上的 CRMd 再将指令转发给当前节点的 LRMd 去处理。当集群节点运行完指令后，运行有 CRMd 进程的其他节点会把他们接收到的全部指令执行结果以及日志返回给 DC（即 DC 最终会收集全部资源在运行集群指令后的结果和状态），然后根据执行结果的实际情况与预期的对比，从而决定当前节点是应该等待之前发起的操作执行完成再进行下一步的操作，还是直接取消当前执行的操作并要求 PEngine 根据实际执行结果再重新规划集群的理想状态并发出操作指令。

在某些情况下，集群可能会要求节点关闭电源以保证共享数据和资源恢复的完整性，为此，Pacemaker 引入了节点隔离机制，而隔离机制主要通过 STONITH 进程实现。STONITH 是一种强制性的隔离措施，STONITH 功能通常是依靠控制远程电源开关以关闭或开启节点来实现。在 Pacemaker 中，STONITH 设备被当成资源模块并被配置到集群信息 CIB 中，从而使其故障情况能够被轻易地监控到。同时，STONITH 进程（STONITHd）能够很好地理解 STONITH 设备的拓扑情况，因此，当集群管理器要隔离某个节点时，只需 STONITHd 的客户端简单地发出 Fencing 某个节点请求，STONITHd 就会自动完成全部剩下的工作，即配置成为集群资源的 STONITH 设备最终便会响应这个请求，并对节点做出 Fenceing 操作，而在实际使用中，根据不同厂商的服务器类型以及节点是物理机还是虚拟机，用户需要选择不同的 STONITH 设备。

3.5 Pacemaker 集群配置信息管理

Pacemaker 集群是通过 CIB 以 XML 的形式进行定义的，而 CIB 主要由集群配置信息与集群状态信息两大部分构成。在 Pacemaker 集群中，未进行任何配置（初始集群）的集群拥有最简单的 CIB，初始集群的 CIB 信息输出如下：

```
<cib crm_feature_set="3.0.7" validate-with="pacemaker-1.2" admin_epoch="1" epoch="0"
  num_updates="0">
  <configuration>
    <crm_config/>
    <nodes/>
    <resources/>
    <constraints/>
  </configuration>
```

```
<status/>
</cib>
```

上述初始集群的 CIB 输出信息中包含了构成 CIB 的主要模块，其中开始和末尾的 cib 标记表明中间内容为集群的 CIB 信息，而 CIB 中的主要内容又分为配置段（configuration 标记）和状态段（status 标记），同时配置段又分为 crm_config、nodes、resources、constraints 四个部分。CIB 中的配置段主要包含当前集群的配置信息，是 CIB 中最为核心的信息，该配置段的信息直接决定了当前集群的资源配置以及集群所能提供的服务，并决定了这些服务彼此之间的联系，以及服务与节点之间的约束和限制。而 CIB 中的状态信息段主要包含有集群当前的资源运行状态信息，状态信息直接反应了当前集群的运行情况，通常而言，CIB 中的集群状态信息主要取决于集群配置信息。

3.5.1 Pacemaker 集群状态信息

集群状态信息包含了集群中每个节点所运行资源的历史信息，根据这些资源的历史数据，集群 PE 将会规划出集群下一阶段应该实现的最理想状态。集群状态信息源自每个节点上的本地资源管理器进程（LRMd），集群运行状态信息会在运行时动态刷新，因此集群不会将状态信息永久性写入磁盘进行保存（这对集群而言并无意义），同时也不建议管理员手动更改集群状态信息，因为集群状态信息的变更应该由集群资源管理器自动刷新。

在 Pacemaker 集群中，查看集群状态信息的工具是 CRM_MON，CRM_MON 是一个用于显示活动 Pacemaker 集群当前状态信息的命令行工具，利用此工具可以通过不同的模式来显示集群的各种状态信息。用户可以在限定节点或资源的前提下运行 CRM_MON 命令行工具，而其集群信息的输出既可以是静态单次模式，也可以是动态刷新模式，CRM_MON 的输出信息即包含以节点和资源为组执行的操作列表，也包含资源运行失败的相关信息。使用 CRM_MON 工具，用户可以检测到集群中的非法操作所引起的集群状态变化，同时还可以通过 CRM_MON 工具进行集群故障仿真等验证性的操作，关于 CRM_MON 的详细使用信息可以通过 `crm_mon --help` 命令进行查看。

在 CRM_MON 命令中，通过不同的 mode 参数和 options 参数组合，用户可以将 Pacemaker 集群当前状态信息以不同的形式输出并进行查看，例如通过 -f 参数可以查看资源运行失败的信息，通过 -h 参数可以将结果以 HTML 的形式输出到指定的文件中，通过 -i 参数可以指定输出结果自动刷新的时间间隔，通过 -l 参数可以将结果定向到标准输出并退出，通过 -o 参数可以查看资源的操作历史等。而在 OpenStack 高可用集群部署中，当集群配置完成并启动资源后，通过简单带有 -f 和 -l 参数的 CRM_MON 命令，便可看到 OpenStack 高可用集群在正常运行下的状态信息，在每个资源都正常运行的情况下，OpenStack 高可用集群的状态信息输出如下：

```
[root@controller1-vm ~]# crm_mon -f -l
```

```
Last updated: Fri Apr 15 16:49:40 2016
```

```
Last change: Sun Apr 10 21:37:31
```

```
2016 by haCluster via crmd on controller3-vm
```



```

Stack: corosync
Current DC: controller3-vm (version 1.1.13-a14efad) - partition with quorum
5 nodes and 231 resources configured
Online: [ controller1-vm controller2-vm controller3-vm ]
RemoteOnline: [ computer1 computer2 ]
    fence1 (stonith:fence_xvm):      Started controller3-vm
    fence2 (stonith:fence_xvm):      Started controller3-vm
    fence3 (stonith:fence_xvm):      Started controller3-vm
    Clone Set: lb-HAproxy-clone [lb-HAproxy]
        Started: [ controller1-vm controller2-vm controller3-vm ]
vip-db (ocf::heartbeat:IPaddr2):      Started controller1-vm
vip-RabbitMQ (ocf::heartbeat:IPaddr2):  Started controller1-vm
vip-keystone (ocf::heartbeat:IPaddr2):  Started controller2-vm
vip-glance (ocf::heartbeat:IPaddr2):    Started controller3-vm
vip-cinder (ocf::heartbeat:IPaddr2):    Started controller1-vm
vip-swift (ocf::heartbeat:IPaddr2):     Started controller2-vm
vip-neutron (ocf::heartbeat:IPaddr2):    Started controller3-vm
vip-nova (ocf::heartbeat:IPaddr2):      Started controller1-vm
vip-horizon (ocf::heartbeat:IPaddr2):    Started controller2-vm
vip-heat (ocf::heartbeat:IPaddr2):      Started controller3-vm
vip-ceilometer (ocf::heartbeat:IPaddr2): Started controller1-vm
vip-qpid (ocf::heartbeat:IPaddr2):      Started controller2-vm
Master/Slave Set: Galera-master [Galera]
    Masters: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: Memcached-clone [Memcached]
    Started: [ controller1-vm controller2-vm controller3-vm ]
.....

```

从 CRM_MON 输出的 OpenStack 高可用集群的当前状态信息中，可以看到集群最近一次状态信息更新的时间和最近一次集群配置变更的时间，还可以看到当前集群的 Stack 是 Corosync（也可以选择 Heatbeat），同时能看到当前集群的 DC 是 controller3-vm 节点。在该集群中，一共配置了 5 个节点（三个本地控制节点和两个远端计算节点）和 231 个资源（以节点为单位进行资源统计）。此外，通过集群状态信息，还能看到 OpenStack 相关的服务在每个节点上的运行情况。当 OpenStack 高可用集群部署完成并正常运行后，全部 OpenStack 相关服务将作为一个完整的 Pacemaker 集群资源而运行，资源之间以及资源与节点之间的关系通过 Pacemaker 的资源约束进行限制，为了排查集群故障和检查 OpenStack 相关服务的运行情况，经常需要通过 CRM_MON 工具来查看集群的运行状态，例如通过 -f 参数来查看当前集群中 OpenStack 相关服务运行失败的历史信息。

3.5.2 Pacemaker 集群配置信息

在 Pacemaker 的 CIB 信息中，除了集群状态信息，集群配置信息也是 Pacemaker 集群中 CIB 信息的关键组成部分，Pacemaker 的集群配置信息决定了集群最终应该如何工作以及集群最终的运行状态，因为只有一个正确的集群配置才能驱动集群资源运行在正常的状态，即正确的配置信息是集群正常运行的前提，因而，从这种因果关系来讲，集群管理员

更应该熟悉集群配置信息。通常情况下，集群的配置信息由集群配置选项（`crm_config`）、集群节点（`nodes`）、集群资源（`resources`）和资源约束（`constraints`）四个配置段组成：

（1）`crm_config`

Pacemaker 集群层面的配置选项都被定义在 `crm_config` 配置段中，`crm_config` 配置段的内容在 CIB 中是以 `<crm_config>` 标记起始并以 `</crm_config>` 标记结尾的配置段，以 Openstack 高可用环境配置为例，Pacemaker 集群 CIB 中的 `crm_config` 配置段信息如下：

```
[root@controller1-vm ~]# cibadmin --query --scope crm_config
<crm_config>
<Cluster_property_set id="cib-bootstrap-options">
<nvpair id="cib-bootstrap-options-have-watchdog" name="have-watchdog"
value="false"/>
<nvpair id="cib-bootstrap-options-dc-version" name="dc-version" value="1.1.13-
a14efad"/>
<nvpair id="cib-bootstrap-options-Cluster-infrastructure" name="Cluster-
infrastructure" value="corosync"/>
<nvpair id="cib-bootstrap-options-Cluster-name" name="Cluster-name"
value="openstack-ha"/>
<nvpair id="cib-bootstrap-options-last-lrm-refresh" name="last-lrm-refresh"
value="1460295450"/>
<nvpair id="cib-bootstrap-options-Cluster-recheck-interval" name="Cluster-
recheck-interval" value="1min"/>
<nvpair id="cib-bootstrap-options-stonith-enabled" name="stonith-enabled"
value="true"/>
</Cluster_property_set>
<Cluster_property_set id="Redis_replication">
<nvpair id="Redis_replication-Redis-server_REPL_INFO" name="Redis-server_REPL_
INFO" value="controller3-vm"/>
</Cluster_property_set>
</crm_config>
```

（2）`nodes`

Pacemaker 集群中的成员节点都被定义在 `nodes` 配置段中，`nodes` 配置段定义了集群的全部节点的 ID 和节点名称，以及节点在集群中的属性等信息，节点配置段在 CIB 中以 `<nodes>` 标记起始并以 `</nodes>` 标记结尾，`nodes` 配置段的内容信息如下：

```
[root@controller1-vm ~]# cibadmin --query --scope nodes
<nodes>
<node id="1" uname="controller1-vm">
<instance_attributes id="nodes-1">
<nvpair id="nodes-1-osprole" name="osprole" value="controller"/>
</instance_attributes>
</node>
<node id="2" uname="controller2-vm">
<instance_attributes id="nodes-2">
<nvpair id="nodes-2-osprole" name="osprole" value="controller"/>
</instance_attributes>
</node>
```

(3) resources

Pacemaker 集群所运行的全部资源都被定义在 Resources 配置段中, 在 Pacemaker 集群的资源配置中, 每个资源都会通过 Class、Type、Provider 等属性对其进行定义, 而这些定义全部位于 Resources 配置段中, 并且每个资源位于一个资源定义段内。资源配置段在 CIB 中以 <resources> 标记起始并以 </resources> 标记结尾, 在 Openstack 等大规模的 Pacemaker 集群中, 资源配置段的内容是非常繁多的, 以 Openstack 集群配置环境为例, Resources 配置段的部分内容信息如下:

```
[root@controller1-vm nova]# cibadmin --query --scope resources
<resources>
.....
<primitive class="ocf" id="vip-glance" provider="heartbeat" type="IPAddr2">
  <instance_attributes id="vip-glance-instance_attributes">
    <nvpair id="vip-glance-instance_attributes-ip" name="ip" value="192.168.142.205"/>
    <nvpair id="vip-glance-instance_attributes-nic" name="nic" value="eth1"/>
  </instance_attributes>
  <operations>
    <op id="vip-glance-start-timeout-20s" interval="0s" name="start" timeout="20s"/>
    <op id="vip-glance-stop-timeout-20s" interval="0s" name="stop" timeout="20s"/>
    <op id="vip-glance-monitor-interval-10s" interval="10s" name="monitor" timeout="20s"/>
  </operations>
</primitive>
.....
</resources>
```



注意 本文中的“.....”均表示省略的标准输出, “\”表示代码换行。

(4) constraints

Pacemaker 集群中各种资源之间的启动顺序及依赖关系等约束都被定义在 Constraints 配置段中, 在由众多资源组成的集群中, 资源之间的启动依赖关系以及资源彼此之间的粘性设置都称为资源约束, 这些约束被统一定义到 Constraints 配置段中, CIB 中的 Constraints 配置段以 <constraints> 标记起始并以 </constraints> 标记结尾, 当资源数量较多, 并且资源之间的约束也较为复杂的情况下, Constraints 配置段的信息量也是相当巨大的, 在我们的 OpenStack 集群配置中, Constraints 配置段的部分内容信息如下:

```
[root@controller1-vm nova]# cibadmin --query --scope constraints
<constraints>
.....
<rsc_order first="vip-db" first-action="start" id="order-vip-db-lb-haproxy-clone-Optional" kind="Optional" then="lb-haproxy-clone" then-action="start"/>
<rsc_colocation id="colocation-vip-db-lb-haproxy-clone-INFINITY" rsc="vip-db" score="INFINITY" with-rsc="lb-haproxy-clone"/>
.....
```

```
<constraints>
```

作为一个集群管理员，经常需要更改或查看集群 CIB 配置信息，而在 Pacemaker 集群中，`cibadmin` 命令行工具是更改和查询集群资源配置信息最为强大的工具。`cibadmin` 命令行工具能够与运行中的 Pacemaker 集群进行实时交互，通过 `cibadmin`，集群管理员可以实时执行集群配置信息的查看、删除、更新或者替换配置信息中的任何部分等操作，并且，使用 `cibadmin` 命令所做的更改会对当前集群立即生效，而无需使用 `reload` 或 `restart` 之类的命令重新加载集群配置。此外，用户通过 `cibadmin` 命令还可以对集群配置信息进行各种自定义的操作，其中，最为常见的 `cibadmin` 操作有以下几种：

```
//查询当前节点的CIB信息
cibadmin --query --local
//仅查询crm_config的配置段信息
cibadmin --query --scope crm_config
//查询全部“target-role”的设置
cibadmin --query --xpath "//nvpair[@name='target-role']"
//删除全部“is-managed”的设置
cibadmin --delete-all --xpath "//nvpair[@name='is-managed']"
//删除资源id为“old”的资源
cibadmin --delete --xml-text '<primitive id="old"/>'
//删除集群全部资源
cibadmin --replace --scope resources --xml-text '<resources/>'
//使用指定文件替换集群全部CIB配置信息
cibadmin --replace --xml-file $HOME/Pacemaker.xml
//使用指定文件替换约束段配置
cibadmin --replace --scope constraints --xml-file $HOME/constraints.xml
//增加配置信息版本以防老版本配置信息被加载
cibadmin --modify --xml-text '<cib admin_epoch="admin_epoch++"/>'
```

对于集群管理员而言，使用 `cibadmin` 最主要的原因之一便是更改集群配置信息，而更改配置信息最常用的方式就是利用 `cibadmin` 将当前集群配置信息保存到一个临时文件中，然后使用自己喜欢的 XML 编辑器对保存配置信息的临时文件进行修改，在确认修改完成之后，再使用 `cibadmin` 命令将修改后的 XML 配置文件替换掉集群的当前配置，而且这种配置信息的替换将会立即生效，使用 `cibadmin` 更新集群配置信息的过程如下所示：

```
//将当前集群配置信息保存到tmp.xml文件中
cibadmin --query > tmp.xml
//使用编辑器对XML文件进行修改（此处使用的是vim）
vim tmp.xml
//将修改后的XML文件替换掉当前集群的配置信息
cibadmin --replace --xml-file tmp.xml
```

Pacemaker 集群的 CIB 配置信息是由 `crm_config` 配置段、`nodes` 配置段、`resources` 配置段、`constraints` 配置段组成的，在大规模集群环境中，通过 `cibadmin` 获取的集群配置信息内容非常繁多，加之 CIB 信息以 XML 语言呈现，这会使得修改最终的 XML 文件非常困难，而在很多时候用户并不需要对 XML 文件的全部内容进行通篇修改，例如用户可能只需

要修改 `crm_config` 配置段中的某个参数或者 `resources` 配置段中的某个资源属性, 这时用户便可仅针对需要的配置段进行配置信息的提取和替换操作, 例如用户需要修改 `resources` 配置段中的某个资源属性, 则可以通过如下方式进行修改替换:

```
cibadmin --query --scope resources > tmp.xml
vim tmp.xml
cibadmin --replace --scope resources --xml-file tmp.xml
```

在大规模的 Pacemaker 集群中, CIB 完整的配置信息可以用海量来形容, 而且全部以 XML 语言来描述, 因此, 尽管有 `cibadmin` 这样的工具存在, 手工修改 CIB 配置也是极为不推荐的。当然, 如果用户确实需要查看 CIB 信息, 那么在正式使用 `cibadmin` 命令之前, 用户首先要清楚定位自己需要的信息应该位于哪个配置段, 并通过 `cibadmin` 命令以配置段为单位进行更为精确的配置信息定位输出, 如下是针对各种配置段的 CIB 信息提取方式:

```
//提取集群全部配置信息
cibadmin --query --local > Cluster.xml
//提取crm_config配置信息段
cibadmin --query --scope crm_config > crm_config.xml
//提取resources配置段
cibadmin --query --scope resources > resources.xml
//提取nodes配置段
cibadmin --query --scope nodes > nodes.xml
//提取constraints配置段
cibadmin --query --scope constraints > constraints.xml
```

另外, 在使用 `cibadmin` 命令更改集群配置信息之前, 建议使用上述命令将集群当前的配置信息导出到临时文件中以便备份集群配置信息, 如果在更改集群配置信息后集群运行状态未能实现预期效果, 则可通过备份文件重新恢复集群配置信息。需要再次说明的是, 通过修改 XML 配置文件来更新集群配置信息是非常麻烦并且特别容易出错的集群变更方式, 尤其是集群节点不断增加并且资源数量非常庞大的情况下, `cibadmin` 输出的 XML 配置文件可能会有成千上万行, 单纯的通过编辑器来修改 XML 文件变得极不现实。因此, 除了 `cibadmin` 命令之外, Pacemaker 还为管理员提供了其他几个非常有用的查看与修改配置信息的命令行工具, 当用户安装 Pacemaker 软件后, 可以到系统的 `/usr/sbin` 目录下找到这些命令:

```
-rwxr-xr-x 1 root root 21592 Sep 1 2015 crmadmin
-rwxr-xr-x 1 root root 21960 Sep 1 2015 crm_attribute
-rwxr-xr-x 1 root root 17064 Sep 1 2015 crm_diff
-rwxr-xr-x 1 root root 11648 Sep 1 2015 crm_error
.....
```

Pacemaker 为这些命令的使用方式准备了详细的说明和使用样例, 用户只需通过 `--help` 参数即可了解这些命令行工具的使用方法, 例如 `crm_attribute` 命令的使用样例可以通过如下方式来获取:

```
[root@controller1-vm sbin]# crm_attribute --help
.....
```



```

Examples:
Add a new attribute called 'location' with the value of 'office' for host 'myhost':
# crm_attribute --node myhost --name location --update office
Query the value of the 'location' node attribute for host myhost:
# crm_attribute --node myhost --name location --query
Change the value of the 'location' node attribute for host myhost:
# crm_attribute --node myhost --name location --update backoffice
Delete the 'location' node attribute for the host myhost:
# crm_attribute --node myhost --name location --delete
Query the value of the Cluster-delay Cluster option:
# crm_attribute --type crm_config --name Cluster-delay --query
Query the value of the Cluster-delay Cluster option. Only print the value:
# crm_attribute --type crm_config --name Cluster-delay --query --quiet

```

例如，用户需要为主机名为 `controller1-vm` 的节点设置一个节点属性，属性名为 `osprole`，属性值为 `controller`，并在设置完成之后对该节点的属性值进行验证，则可以通过 `crm_attribute` 命令来实现：

```

//为controller1-vm节点设置osprole属性
[root@controller1-vm sbin]# crm_attribute --node controller1-vm --name osprole
--update=controller
//查看controller1-vm节点的osprole属性
[root@controller1-vm sbin]# crm_attribute --node controller1-vm --name osprole
--query
//输出信息如下：
scope=nodes name=osprole value=controller

```

如果用户需要查看某个资源的运行情况，则可以通过 `crm_resource` 命令来实现，该命令将会列出该资源在全部集群节点上的运行情况。例如，我们需要查看预定义的 Openstack 网络 API 服务 `neutron-server-api` 资源的节点运行情况，则可以通过如下方式实现：

```

[root@controller1-vm sbin]# crm_resource --resource neutron-server-api --locate
//输出结果如下：
resource neutron-server-api is running on: controller3-vm
resource neutron-server-api is running on: controller1-vm
resource neutron-server-api is running on: controller2-vm
resource neutron-server-api is NOT running
resource neutron-server-api is NOT running

```

上述输出结果说明 `neutron-server-api` 资源同时运行在三个控制节点上，而其他节点并不运行 `neutron-server-api` 资源。此外，为了仿真测试集群资源变更，Pacemaker 还提供了一个仿真集群事件响应的工具 `crm_simulate`，`crm_simulate` 的具体使用可以通过 `--help` 参数查看，其使用语法如下：

```
crm_simulate datasource operation
```

其中，`crm_simulate` 利用数据源（`datasource`）来执行指定的操作（`operation`），仿真之后的结果可以保存为不同的数据格式。在 Linux 系统中，一种便于查看的格式就是保存为

Graphviz 的 dot 类型文件，然后通过 Graphviz 的 dot 命令可以将 dot 文件渲染为关系图，通过关系图中的箭头指向可以比较清晰地看到集群对于配置变更或集群事件的响应过程。例如，在我们的 OpenStack 集群配置中，可以通过 `crm_simulate` 命令来仿真 OpenStack 集群启动过程中集群事件的响应过程和资源之间的启动关系，但是由于 OpenStack 集群资源众多并且资源之间的约束限制错综复杂，导致最终 Graphviz 绘制的图形也极为复杂，图 3-9 仅是仿真结果的冰山一角。`crm_simulate` 命令的使用很简单，只需集合 Graphviz 软件的 dot 命令即可将仿真的关系数据进行绘制，并保存为各种格式的图片：

```
root@controller1-vm ~]# crm_simulate -L -VVVVV \
--save-dotfile=openstack.dot
root@controller1-vm ~]# dot -T png openstack.dot -o openstack.png
```

上述第一条命令通过 `crm_simulate` 工具，利用当前集群 CIB 配置信息来进行仿真，并将结果保存到 `openstack.dot` 文件中（数据源）；第二条命令利用 Graphviz 工具的 dot 命令对 `openstack.dot` 文件进行绘制并保存为 png 格式图像 `openstack.png`。图 3-9 便是关系图 `openstack.png` 的小部分截图，完整的关系图比图 3-9 要复杂得多。通常，`crm_simulate` 命令和 Graphviz 软件生成的关系图是对集群配置信息的直观绘制，相对于庞大复杂的 XML 配置文件，以箭头指向的形式呈现的集群配置信息更容易被理解和掌握，而熟悉集群配置和资源关系的管理人员便可通过这种直观的关系图来判断集群资源的启动运行情况。



图 3-9 OpenStack 集群 CIB 仿真截图

3.6 Pacemaker 集群管理工具 PCS

3.6.1 PCS 命令行工具

3.5 节介绍了用 cibadmin 命令行工具来查看和管理 Pacemaker 的集群配置信息，正如前文所述，集群 CIB 中的配置信息量非常大而且以 XML 语言呈现，对于仅由极少数节点和资源所组成的集群，cibadmin 也许是个可行方案。但是，对于拥有大量节点和资源的大规模集群，通过编辑 XML 文件来查看修改集群配置显然是非常艰难而且极为不现实的工作，由于 XML 文件内容条目极多，因此用户在修改 XML 文件的过程中极易出现人为错误。而在开源社区里，简单实用才是真正开源精神的体现，对于开源系统中任何文件配置参数的修改，简化统一的命令行工具才是最终的归宿。

随着开源集群软件 Pacemaker 版本的不断更新，社区推出了两个常用的集群管理命令行工具，即集群管理员最为常用的 pcs 和 crmsh 命令。本文使用的是 pcs 命令行工具，关于 crmsh 的更多使用方法和手册可以参考 Pacemaker 的官方网站。在 Pacemaker 集群中，pcs 命令行工具几乎可以实现集群管理的各种功能，例如，全部受控的 Pacemaker 和 Corosync 配置属性的变更管理都可以通过 pcs 实现。此外，需要注意的是，pcs 命令行的使用对系统中安装的 Pacemaker 和 Corosync 软件版本有一定要求，即 Pacemaker1.1.8 及其以上版本，Corosync2.0 及其以上版本才能使用 pcs 命令行工具进行集群管理。pcs 命令可以管理的集群对象类别和具体使用方式可以通过 --help 参数查看：

```
[root@ controller1-vm ~]# pcs --help
Usage: pcs [-f file] [-h] [commands]...
Control and configure Pacemaker and corosync.
Options:
    -h, --help      Display usage and exit
    -f file          Perform actions on file instead of Active CIB
    --debug          Print all network traffic and external commands run
    --version        Print pcs version information

Commands:
Cluster      Configure Cluster options and nodes
resource     Manage Cluster resources
stonith      Configure fence devices
constraint   Set resource constraints
property     Set Pacemaker properties
acl          Set Pacemaker access control lists
status       View Cluster status
config       View and manage Cluster configuration
```

pcs 将集群划分为不同的管理对象，针对某个管理对象有不同的命令集可以使用。通过 --help 参数可以看到 pcs 可管理的集群对象种类包括 Cluster、Resource、Stonith、Constraint、Property、Status、ACL 和 Config 等，其中最为常用的管理类别有以下几个。

- ❑ cluster：配置集群选项和节点。
- ❑ status：查看当前集群资源和节点以及进程的状态。

- ❑ resource: 创建和管理集群资源。
- ❑ constraint: 管理集群资源约束和限制。
- ❑ property: 管理集群节点和资源属性。
- ❑ config: 以用户可读格式显示完整集群配置信息。

要查看 pcs 针对不同集群对象类别的管理命令，可以通过 `pcs category_name help` 命令（“category_name”表示类别名称）来查看，如要查看 pcs 命令对 Status 管理类别的命令使用方法，可以通过如下方式实现：

```
[root@ controller1-vm ~]# pcs status help
Usage: pcs status [commands]...
View current Cluster and resource status
Commands:
    [status] [--full]
        View all information about the Cluster and resources (--full provides
more details)
resources
    View current status of Cluster resources
groups
    View currently configured groups and their resources
Cluster
    View current Cluster status
corosync
    View current membership information as seen by corosync
nodes [corosync|both|config]
    View current status of nodes from Pacemaker. If 'corosync' is
specified, print nodes currently configured in corosync, if 'both'
is specified, print nodes from both corosync & Pacemaker. If 'config'
is specified, print nodes from corosync & Pacemaker configuration.
pcsd <node> ...
    Show the current status of pcsd on the specified nodes
xml
    View xml version of status (output from crm_mon -r -l -X)
```

pcs 的 help 命令提供了在该管理类别中，pcs 能够实现的全部子命令的详细使用说明，从 pcs 对 Status 类别的帮助文档中可以看到，pcs 不仅可以查看当前集群的状态，还可以查看集群资源的状态。例如，在 OpenStack 的高可用部署环境中，我们经常需要通过 pcs 命令检查当前集群的资源运行情况，从而判断哪些资源正常运行，哪些资源运行异常，而对于异常的资源则需要做进一步的处理（例如通过 `clearup` 或 `restart` 进行资源重启）。在 OpenStack 高可用集群资源部署完成后，可以通过 `pcs status` 命令检查集群中的资源是否与预期规划一致，如下：

```
root@controller1-vm ~]# pcs status resources
.....
Clone Set: libvirt-compute-clone [libvirt-compute]
Started: [ computer1 computer2 ]
Clone Set: ceilometer-compute-clone [ceilometer-compute]
```

```

Started: [ computer1 computer2 ]
Clone Set: nova-compute-fs-clone [nova-compute-fs]
Started: [ computer1 computer2 ]
fence-nova      (stonith:fence_compute): Started controller2-vm
computer1      (ocf::Pacemaker:remote): Started controller2-vm
computer2      (ocf::Pacemaker:remote): Started controller3-vm
Clone Set: nova-compute-clone [nova-compute]
Started: [ computer1 computer2 ]

```

从 pcs status resource 的输出中，我们可以清楚地看到当前 OpenStack 集群中有哪些资源存在，并且可以看到每个资源（OpenStack 相关服务）在节点上的运行状态。通过 pcs 的 status 命令，用户除了可以查看 cluster 和 resource 的状态，还可以查看集群节点在集群中的运行状态，即哪些节点当前在线，哪些节点已经离线，如下：

```

[root@controller1-vm ~]# pcs status nodes
Pacemaker Nodes:
    Online: controller1-vm controller2-vm controller3-vm
    Standby:
    Offline: computer1 computer2

```

在 Pacemaker 集群的介绍中，我们曾经提到 Corosync 的主要作用之一便是向 Pacemaker 集群提供节点之间的成员关系（Membership），通过 pcs 命令，Corosync 所提供的 Membership 便清晰可见：

```

[root@controller1-vm ~]# pcs status corosync
Membership information

```

Nodeid	Votes	Name
1	1	controller1-vm (local)
2	1	controller2-vm
3	1	controller3-vm

除了 Status 管理类别，pcs 经常使用的管理类别还包括 Cluster、Resource、Stonith、Constraint、Property 和 Config，而每个管理类别的使用方式都可以通过 help 参数查看，通过对这几个集群类别的管理配置，用户便可以配置出具有很多高级功能的 Pacemaker 集群。关于 Config 类别，3.5 节中曾经使用 cibadmin 命令来进行集群配置的备份和恢复工作，其实 pcs 所提供的 config 管理类别也能实现相同的功能，例如要保存当前集群的配置信息，则在 pcs 的 config 管理类别中执行 backup 命令即可。

```

//语法格式为：pcs config backup file_name
# pcs config backup openstack_ha
//上述命令会生成一个file_name.tar.bz2格式的备份文件：
# ls -l openstack_ha*
-rw-r--r-- 1 root root 5934 Apr 16 23:04 openstack_ha.tar.bz2

```

3.6.2 PCS 用户接口界面

除了命令行管理配置工具，pcs 还提供了交互式的页面管理界面（GUI），用户通过 GUI

便可直观地对 Pacemaker 集群进行配置和监测。要使用 pcs 工具的 GUI 功能，只需经过以下几个配置步骤即可：

1) 安装 Pacemaker1.1.8 和 Corosync2.0 及其以上版本，同时安装 pcs 配置工具。

2) 在 Pacemaker 集群的每个节点中使用 passwd 命令为用户 haCluster 设定密码（安装集群软件时自动创建的用户），在全部集群节点中为该用户设置相同的密码。

3) 在 Pacemaker 集群的每个节点中均启动 pcsd 守护进程：

```
systemctl start pcsd.service
systemctl enable pcsd.service
```

4) 在 Pacemaker 集群的每个节点中使用以下命令将节点认证到集群中，运行 auth 命令后系统会提示用户输入 username 和 password，将此处的 username 指定为 haCluster，密码为第 2 步中设置的密码：

```
pcs cluster auth node1 node2 . . . nodeN
```

5) 在与 Pacemaker 集群相同网段的任意服务器上打开浏览器，输入 pcsd 守护进程的监听地址及端口即可访问 pcs 的 GUI。因为每个节点都有 pcsd 进程运行，并且每个节点都在集群中进行了认证，因此可以任意指定某个 Pacemaker 节点的 IP 地址，一旦输入正确的 IP 地址和端口，即可显示 pcs 的 GUI 用户登录页面。例如在浏览器中输入 https://192.168.142.110:2224，其中 192.168.142.110 为 controller1-vm 节点 pcsd 进程的监听 IP 地址，2224 为 pcsd 默认的监听端口，在浏览器地址栏输入如上地址后，将会看到如图 3-10 所示的登录界面：

图 3-10 Pacemaker 集群管理工具 GUI 登录界面

6) 使用 haCluster 用户和对应的密码即可登录 GUI 页面并进行 Pacemaker 集群的配置和查看，例如在我们的五节点 OpenStack 示例环境中，登录 GUI 之后将会看到如图 3-11 所示的界面：

进入 pcs 集群管理工具的 GUI 之后，用户便可通过 GUI 的管理集群（Manage Clusters）菜单页面进行集群创建或者添加现有集群等操作。在用户成功创建集群后，Manage Clusters 页面中会显示该集群的名称，只需将光标移动到该集群名称上即可显示该集群的详细信息。

□ 新建集群。在 ManageClusters 页面中单击新建按钮，并输入要创建的集群名称以及组成该集群的节点名称，输入全部信息后，单击创建集群，创建成功后会在 ManageClusters 页面中显示刚刚创建的集群及该集群中的节点信息。

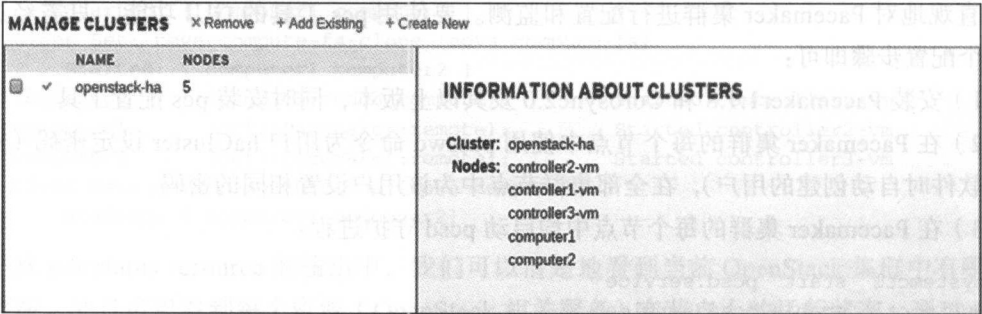


图 3-11 Pacemaker 集群管理工具 GUI 界面

❑ 添加现有集群。Pacemaker 的 pcs 集群管理工具 GUI 可以管理已有的集群，要管理已有集群，只需进行添加操作。在 ManageClusters 页面中单击添加现有集群按钮，并输入要使用 GUI 管理的集群节点主机名和 IP 地址即可。

要管理某个集群，只需选中并单击集群名称，单击集群名称后会出现一个集群管理配置页面，在该页面中即可配置该集群的节点、资源、Fencing 设备和集群属性。要查看某个节点的运行情况，只需选中并单击该节点名称，便可以看到该节点上的 Pacemaker 和 Corosync 以及 pcsd 进程的运行情况，同时还可以对该节点进行 start、stop 和 standby 等操作，此外，节点页面还会显示该节点上运行的全部资源，如图 3-12 所示。

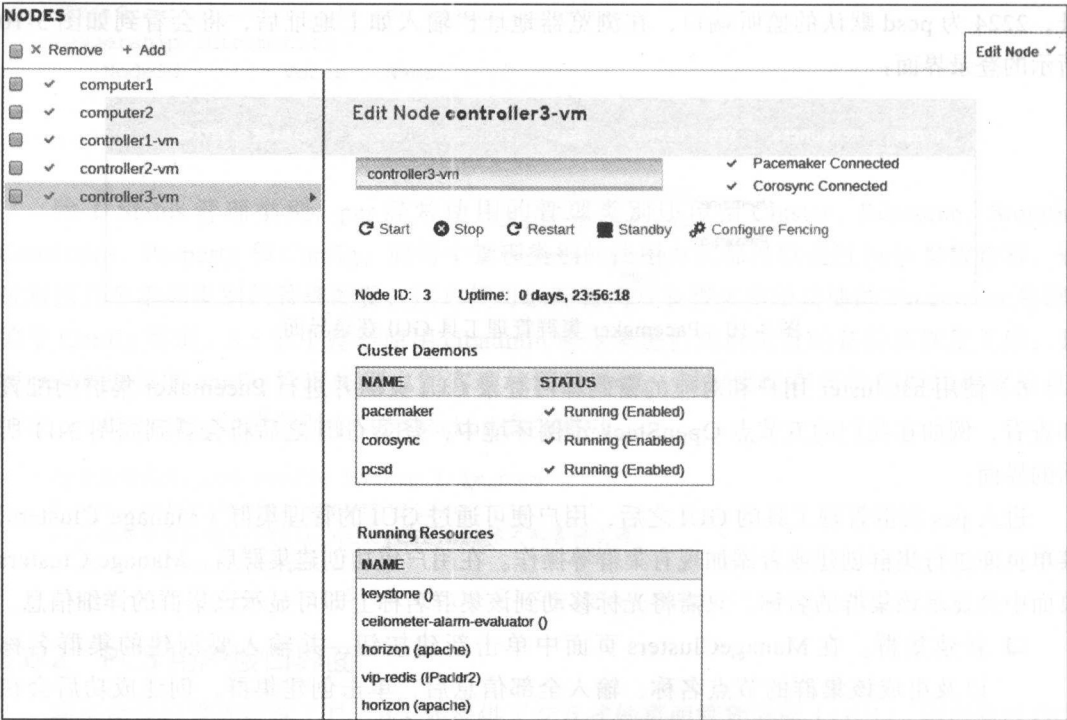


图 3-12 Pacemaker 管理工具 GUI 节点管理页面

在 Pacemaker 的集群配置中, 通过 pcs 命令行能够实现的集群管理配置操作, 通过 pcsd 守护进程提供的 Web 界面同样可以实现。对于不熟悉 pcs 命令行的管理员而言, pcsd 的 GUI 管理页面提供了直观简洁的集群配置方式, 同时 GUI 还提供了集中管理多个 Pacemaker 集群的功能, 这对于运行大量集群的系统, 将会极大简化运行维护的工作任务。

3.7 Pacemaker 集群资源管理

3.7.1 集群资源代理

在 Pacemaker 高可用集群中, 资源就是集群所维护的高可用服务对象。根据用户的配置, 资源有不同的种类, 其中最为简单的资源是原始资源 (Primitive Resource), 此外还有相对高级和复杂的资源组 (Resource Group) 和克隆资源 (Clone Resource) 等集群资源概念。在 Pacemaker 集群中, 每一个原始资源都有一个资源代理 (Resource Agent, RA), RA 是一个与资源相关的外部脚本程序, 该程序抽象了资源本身所提供的服务并向集群呈现一致的视图以供集群对该资源进行操作控制。通过 RA, 几乎任何应用程序都可以成为 Pacemaker 集群的资源从而被集群资源管理器和控制。RA 的存在, 使得集群资源管理器可以对其所管理的资源“不求甚解”, 即集群资源管理器无需知道资源具体的工作逻辑和原理 (RA 已将其封装), 资源管理器只需向 RA 发出 Start、Stop、Monitor 等命令, RA 便会执行相应的操作。从资源管理器对资源的控制过程来看, 集群对资源的管理完全依赖于该资源所提供的 RA, 即资源的 RA 脚本功能直接决定了资源管理器可以对该资源进行何种控制, 因此一个功能完善的 RA 在发行之前必须经过充分的功能测试。在多数情况下, 资源 RA 以 shell 脚本的形式提供, 当然也可以使用其他比较流行的如 C、Python、Perl 等语言来实现 RA。

在 Pacemaker 集群中, 资源管理器支持不同种类的资源代理, 这些受支持的资源代理包括 OCF、LSB、Upstart、Systemd、Service、Fencing、Nagios Plugins。而在 Linux 系统中, 最为常见的有 OCF (Open Cluster Framework) 资源代理、LSB (Linux Standard Base) 资源代理、Systemd 和 Service 资源代理。

(1) OCF

OCF 是开放式集群框架的简称, 从本质上来看, OCF 标准其实是对 LSB 标准约定中 init 脚本的一种延伸和扩展。OCF 标准支持参数传递、自我功能描述以及可扩展性, 此外, OCF 标准还严格定义了操作执行后的返回代码, 集群资源管理器将会根据 OCF 资源代理返回的执行代码来对执行结果做出判断。因此, 如果 OCF 脚本错误地提供了与操作结果不匹配的返回代码, 则执行操作后的集群资源行为可能会变得莫名其妙, 而对于不熟悉 OCF 脚本的用户, 这将会是个非常困惑和不解的问题, 尤其是当集群依赖于 OCF 返回代码来在资源的完全停止状态、错误状态和不确定状态之间进行判断的时候。因此, 在 OCF 脚本发行使用之前一定要经过充分的功能测试, 否则有问题的 OCF 脚本将会扰乱整个集群的资源管

理。在 Pacemaker 集群中，OCF 作为一种可以自我描述和高度灵活的行业标准，其已经成为使用最多的资源类别。

(2) LSB

LSB 是最为传统的 Linux 资源标准之一，例如在 Redhat 的 RHEL6 及其以下版本中（或者对应的 CentOS 版本中），经常在 `/etc /init.d` 目录下看到的资源启动脚本便是 LSB 标准的资源控制脚本。通常，LSB 类型的脚本是由操作系统的发行版本提供的，而为了让集群能够使用这些脚本，它们必须遵循 LSB 的规定，LSB 类型的资源是可以配置为系统启动时自动启动的，但是如果需要通过集群资源管理器来控制这些资源，则不能将其配置为自动启动，而是由集群根据策略来自行启动。

(3) Systemd

在很多 Linux 的最新发行版本中，Systemd 被用以替换传统“SysV”风格的系统启动初始化进程和脚本，如在 Redhat 的 RHEL7 和对应的 CentOS7 操作系统中，Systemd 已经完全替代了 Sysvinit 启动系统，同时 Systemd 提供了与 Sysvinit 以及 LSB 风格脚本兼容的特性，因此老旧系统中已经存在的服务和进程无需修改便可使用 Systemd。在 Systemd 中，服务不再是 `/etc/init.d` 目录下的 shell 脚本，而是一个单元文件（unit-file），Systemd 通过单元文件来启停和控制服务，Pacemaker 提供了管理 Systemd 类型的应用服务的功能。

(4) Service

Service 是 Pacemaker 支持的一种特别的服务别名，由于系统中存在各种类型的服务（如 LSB、Systemd 和 OCF），Pacemaker 使用服务别名的方式自动识别在指定的集群节点上应该使用哪一种类型的服务。当一个集群中混合有 Systemd、LSB 和 OCF 类型资源的时候，Service 类型的资源代理别名就变得非常有用，例如在存在多种资源类别的情况下，Pacemaker 将会自动按照 LSB、Systemd、Upstart 的顺序来查找启动资源的脚本。

在 Pacemaker 中，每个资源都具有属性，资源属性决定了该资源 RA 脚本的位置，以及该资源隶属于哪种资源标准。例如，在某些情况下，用户可能会在同一系统中安装不同版本或者不同来源的同一服务（如相同的 RabbitMQ Cluster 安装程序可能来自 RabbitMQ 官方社区也可能来自 Redhat 提供的 RabbitMQ 安装包），在这个时候，就会存在同一服务对应多个版本资源的情况，为了区分不同来源的资源，就需要在定义集群资源的时候通过资源属性来指定具体使用哪个资源。在 Pacemaker 集群中，资源属性由以下几个部分构成。

- ❑ Resource_id: 用户定义的资源名称。
- ❑ Standard: 脚本遵循的标准，允许值为 OCF、Service、Upstart、Systemd、LSB、Stonith。
- ❑ Type: 资源代理的名称，如常见的 IPAddr 便是资源的 Type。
- ❑ Provider: OCF 规范允许多个供应商提供同一资源代理，Provider 即是指资源脚本的提供者，大多数 OCF 规范提供的资源代理均使用 Heartbeat 作为 Provider。

上述资源属性在集群定义资源时候均会使用，例如要在集群中创建一个名称为 neutron-

ovs-cleanup 的资源 (resource_id 属性), 并指定资源的 RA 符合 OCF 规范 (standard 属性), 该 RA 由 OpenStack 的 Neutron 组件提供 (Provider 属性), RA 的名称为 OVSCleanup (Type 属性), 通过 PCS 命令行工具的 Resource 命令来创建资源, 则最终的资源定义语句如下:

```
pcs resource create neutron-ovs-cleanup ocf:neutron:OVSCleanup --clone\ interleave=true
```

通过上述语句定义资源后, 集群资源管理器将根据 Standard、Provider 以及 Type 属性信息对 neutron-ovs-cleanup 资源对应的 RA 脚本位置进行定位, 在 RHEL 或者 Centos 系统中, 软件安装完成后, 符合 OCF 规范的 RA 脚本均位于 /usr/lib/ocf/resource.d 目录中。例如, 在成功安装 OpenStack 的 Neutron 组件后, 便可以到该目录下找到 Type 为 OVSCleanup 的 RA 脚本:

```
[root@controller1-vm ~]# cd /usr/lib/ocf/resource.d && ll
drwxr-xr-x 2 root root 4096 Mar  2 17:31 heartbeat
drwxr-xr-x 2 root root  61 Feb  8 02:55 neutron
drwxr-xr-x 2 root root  43 Mar  2 14:44 openstack
drwxr-xr-x 2 root root 4096 Mar  2 14:52 Pacemaker
drwxr-xr-x 2 root root 119 Mar  5 21:37 RabbitMQ
```

该目录下存放了全部的 Providers, 每个 Provider 目录下会有多个 Types。在该示例中, /usr/lib/ocf/resource.d 目录有 5 个子目录, 每个子目录均是一个 Provider, 其中便有名为 neutron 的 Provider, 该 Provider 中有如下的 Type:

```
[root@controller1-vm resource.d]# cd neutron && ll
-rwxr-xr-x 1 root root 4640 Apr 25 2015 NetnsCleanup
-rwxr-xr-x 1 root root 5831 Apr 26 2015 NeutronScale
-rwxr-xr-x 1 root root 4606 Apr 25 2015 OVSCleanup
```

可以看到, /usr/lib/ocf/resource.d/neutron 目录下有 3 个可执行文件, 每一个就是一个 Type (即 RA 脚本的名称), 在此, 除了 OVSCleanup 外, neutron 这个 Provider 还提供了 NeutronScale 和 NetnsCleanup 两个 Type。在创建 neutron-ovs-cleanup 资源的过程中, 定义资源属性的语句段是 ocf:neutron:OVSCleanup (standard:provider:type), 通过该信息, 集群资源管理器便可定位资源 RA 脚本的最终位置, 并使用该脚本来控制资源。资源属性的查看可以通过 pcs 命令行工具实现, 最常使用的资源属性查看方式有以下几种:

- ❑ pcs resource list: 查看集群中所有可用资源列表。
- ❑ pcs resource standards: 查看支持的资源代理标准。
- ❑ pcs resource providers: 查看集群中可用资源代理提供程序列表。
- ❑ pcs resource list string: 查看根据指定字符串过滤的可用资源列表, 可使用这个命令根据名称、提供程序或类型过滤资源。
- ❑ pcs resource describe standard:provider:type: 查看 standard:provider:type 指定的资源代理的详细信息。

如下示例演示了集群资源属性的查看方式及结果, 示例环境为 CentOS7 系统, 系统中

已经安装有 OpenStack 相关的软件包和 Pacemaker 集群软件：

```
//查看当前集群中支持哪些标准(standards)
[root@controller1-vm ~]# pcs resource standards
ocf
lsb
service
systemd
stonith
//查看当前集群中有哪些资源代理提供程序(providers)
[root@controller1-vm ~]# pcs resource providers
heartbeat
neutron
openstack
Pacemaker
RabbitMQ
//查看provider为heartbeat的提供者提供了哪些资源代理(Types)
[root@controller1-vm ~]# pcs resource list heartbeat
ocf:heartbeat:CTDB - CTDB Resource Agent
ocf:heartbeat:Delay - Waits for a defined timespan
ocf:heartbeat:Dummy - Example stateless resource agent
ocf:heartbeat:Filesystem - Manages filesystem mounts
ocf:heartbeat:docker - Docker container resource agent.
ocf:heartbeat:MySQL - Manages a MySQL database instance
ocf:heartbeat:RabbitMQ-Cluster - RabbitMQ Clustered
.....
//查看openstack这个provider提供了哪些资源代理(RA)
[root@controller1-vm ~]# pcs resource list openstack
ocf:openstack:NovaCompute - OpenStack Nova Compute Server
ocf:openstack:NovaEvacuate - Evacuator for OpenStack Nova Compute Server.
systemd:openstack-ceilometer-alarm-evaluator
systemd:openstack-ceilometer-alarm-notifier
systemd:openstack-ceilometer-api
systemd:openstack-ceilometer-central
systemd:openstack-ceilometer-collector
.....
```

从示例输出中可以看到，当前集群支持的规范包括 OCF、LSB、Service、Systemd 和 Stonith，而集群中的 Provides 包括 Heartbeat、Neutron、Openstack、Pacemaker、RabbitMQ 等，符合 OCF 规范的 Heartbeat 资源代理供应程序提供了包括 MySQL、RabbitMQ-Cluster、IPAddr2 和 Filesystem 等资源代理。其中，OpenStack 这个 Provider 提供了符合 OCF 和 Systemd 两种规范的资源代理，这些资源代理是 OpenStack 高可用集群部署的基础，Pacemaker 集群将通过这些资源代理对 OpenStack 服务进行控制。在众多资源代理中，用户可能想要了解某个资源代理的具体信息和使用方法，例如要了解 NovaCompute 资源代理的介绍和使用参数等信息，则可以通过如下命令实现：

```
[root@controller1-vm ~]# pcs resource describe ocf:openstack:NovaCompute
ocf:openstack:NovaCompute - OpenStack Nova Compute Server
```

OpenStack Nova Compute Server.

Resource options:

auth_url (required): Authorization URL for connecting to keystone in admin context

username (required): Username for connecting to keystone in admin context

password (required): Password for connecting to keystone in admin context

tenant_name (required): Tenant name for connecting to keystone in admin context.

Note that with Keystone V3 tenant names are only unique within a domain.

domain: DNS domain in which hosts live, useful when the Cluster uses short names and nova uses FQDN

endpoint_type: Nova API location (internal, public or admin URL)

no_shared_storage: Disable shared storage recovery for instances. Use at your own risk!

evacuation_delay: How long to wait for nova to finish evacuating instances elsewhere before starting nova-compute. Only used when the agent detects evacuations might be in progress. You may need to increase the start timeout when increasing this value.

`pcs resource` 命令的 `describe` 选项将会输出资源代理的详细使用信息，包括该资源代理的介绍和可以传递进去的参数。此外，由于不同的规范和 `Provider` 可以提供相同名称的资源代理，因此在一个集群系统中可能包含有多个同名的资源代理，如下：

```
[root@controller1-vm ~]# pcs resource agents|grep rabbit
RabbitMQ-Cluster
RabbitMQ-server
RabbitMQ-server-ha
set_RabbitMQ_policy.sh
RabbitMQ-server
```

虽然集群中有同名的资源代理（如 `RabbitMQ-server`），但是由于它们属于不同的规范或者相同规范下不同的 `Provider`，所以当资源管理器在调用这些同名的资源代理对资源进行管理时，操作执行后返回的结果可能完全不一样。为此，类似情况下，在定义集群资源的时候必须指定资源代理的绝对路径，从而告诉资源管理器在对该资源进行管理控制时具体调用的资源代理脚本在哪里。在 `Pacemaker` 集群中，要获取同名资源代理的绝对路径，可以通过如下方式实现：

```
[root@controller1-vm ~]# pcs resource list RabbitMQ-server
ocf:RabbitMQ:RabbitMQ-server - Resource agent for RabbitMQ-server
lsb:RabbitMQ-server
```

示例输出中有两个同名的资源代理 `RabbitMQ-sever`，其中一个属于 `LSB` 规范，另一个属于 `OCF` 规范，因此，从本质上来说这是两个完全不同的资源代理（来自不同版本或发行方的软件安装）。在 `Pacemaker` 集群中创建 `RabbitMQ` 资源时，通过 `lsb:rabitmq-sever` 告知资源管理器使用的是符合 `LSB` 规范的 `RabbitMQ` 资源代理，通过 `ocf:RabbitMQ:RabbitMQ-sever` 则告知资源管理器使用的是符合 `OCF` 规范的 `RabbitMQ` 资源代理。此外，多数 `OCF` 规范的资源代理都有资源选项（`Resource Options`）和元数据选项（`Meta`），资源选项即是可以传递给该资源代理的参数，`Resource Options` 和 `Meta` 可以在创建资源时指定，也可以在

资源创建完成后进行修改, 创建带有 Resource Options 和 Meta 选项资源的语法如下:

```
pcs resource create resource_id standard:provider:type[type [resource options]
[meta meta_options...]]
```

如下命令在集群中创建一个名为 vip-qpid 的资源, 并在资源选项中设定了该资源的 IP 地址和子网掩码, 同时指定元数据 Resource-Stickiness 的值为 50:

```
pcs resource create vip-qpid ocf:heartbeat:IPaddr2 ip= 192.168.0.120 cidr_net-
mask=24 meta resource-stickiness= 50
```

元数据 Resource-Stickiness 表示资源的粘性, 即资源倾向于当前状态的程度。如果资源已经创建完成, 因为系统参数调整而要修改资源参数, 例如集群中已经存在资源 vip-qpid, 现要修改资源的 IP 地址和 Resource-Stickiness 的值, 则可以通过如下命令实现:

```
//修改前的vip-qpid资源属性如下:
[root@controller1-vm ~]# pcs resource show vip-qpid
Resource: vip-qpid (class=ocf provider=heartbeat type=IPaddr2)
Attributes: ip=192.168.0.120 nic=eth1
Meta Attrs: resource-stickiness=50
Operations: start interval=0s timeout=20s
//修改元数据Resource-Stickiness
[root@controller1-vm ~]# pcs resource meta vip-qpid\
resource-stickiness=INFINITY
//修改资源选项IP地址值
[root@controller1-vm ~]# pcs resource update vip-qpid ip=192.168.142.215
//修改后的vip-qpid资源属性如下:
[root@controller1-vm ~]# pcs resource show vip-qpid
Resource: vip-qpid (class=ocf provider=heartbeat type=IPaddr2)
Attributes: ip=192.168.142.215 nic=eth1
Meta Attrs: resource-stickiness=INFINITY
Operations: start interval=0s timeout=20s
```

在 Pacemaker 集群中, 另一个常见的与资源管理相关的操作, 就是重置资源状态并清除资源失败计数器, 资源状态重置后, 将会触发集群重新启动资源, 这对于重新启动因处于失败状态而失去资源管理器控制的资源很有用, 重置资源状态的语法如下:

```
pcs resource cleanup resource_id
```

如果在命令中没有指定 Resource_id, 则这个命令会对集群中全部的资源进行状态重置和失败计数器清零, 并且按照资源的启动依赖关系重新启动集群的全部资源, 在集群资源管理中, 在需要重启全部集群资源时, 经常使用该方法。

3.7.2 集群资源约束

集群是由众多具有特定功能的资源组成的集合, 集群中的每个资源都可以对外提供独立服务, 但是资源彼此之间又存在着依赖与被依赖的关系。如资源 B 的启动必须依赖资源 A 的存在, 因此资源 A 必须在资源 B 之前启动, 再如资源 A 必须与资源 B 位于同一节点

以共享某些服务，则资源 B 与 A 在故障切换时必须作为一个逻辑整体而同时迁移到其他节点，在 Pacemaker 中，资源之间的这种关系通过资源约束或限制（Resource Constraint）来实现。Pacemaker 集群中的资源约束可以分为以下几类。

- ❑ 位置约束（Location）：位置约束限定了资源应该在哪个集群节点上启动运行。
- ❑ 顺序约束（Order）：顺序约束限定了资源之间的启动顺序。
- ❑ 资源捆绑约束（Colocation）：捆绑约束将不同的资源捆绑在一起作为一个逻辑整体，即资源 A 位于 C 节点，则资源 B 也必须位于 C 节点，并且资源 A、B 将会同时进行故障切换到相同的节点上。

在资源配置中，Location 约束在限定运行资源的节点时非常有用，例如在 Openstack 高可用集群配置中，我们希望 Nova-Compute 资源仅运行在计算节点上，而 Nova-api 和 Neutron-server 等资源仅运行在控制节点上，这时便可通过资源的 Location 约束来实现。例如，我们先给每一个节点设置不同的 osprole 属性（属性名称可自定义），计算节点中该值设为 compute，控制节点中该值设为 controller，如下：

```
pcs property set --node compute1 osprole=compute
pcs property set --node compute1 osprole=compute
pcs property set --node controller1-vm osprole=controller
pcs property set --node controller2-vm osprole=controller
pcs property set --node controller3-vm osprole=controller
```

然后，通过为资源设置 Location 约束，便可将 Nova-Compute 资源仅限制在计算节点上运行，Location 约束的设置命令如下：

```
pcs constraint location nova-compute-clone rule\ resource-discovery=exclusive
score=0 osprole eq compute
```

即资源 Nova-Compute-Clone 仅会在 osprole 等于 compute 的节点上运行，也即计算节点上运行。

在 Pacemaker 集群中，Order 约束主要用来解决资源的启动依赖关系，资源启动依赖在 Linux 系统中非常普遍。例如在 OpenStack 高可用集群配置中，需要先启动基础服务如 RabbitMQ 和 MySQL 等，才能启动 OpenStack 的核心服务，因为这些服务都需要使用消息队列和数据库服务；再如在网络服务 Neutron 中，必须先启动 Neutron-server 服务，才能启动 Neutron 的其他 Agent 服务，因为这些 Agent 在启动时均会到 Neutron-server 中进行服务注册。Pacemaker 集群中解决资源启动依赖的方案便是 Order 约束。例如，在 OpenStack 的网络服务 Neutron 配置中，与 Neutron 相关的资源启动顺序应该如下：Keystone--> Neutron-server--> Neutron-ovs-cleanup-->Neutron-netns-cleanup-->Neutron-openvswitch-agent-->Neutron-dhcp-agent-->Neutron-l3-agent，上述依赖关系可以通过如下 Order 约束实现：

```
pcs constraint order start keystone-clone then neutron-server-api-clone
pcs constraint order start neutron-server-api-clone then neutron-ovs-cleanup-clone
pcs constraint order start neutron-ovs-cleanup -clone then neutron-netns-cleanup-clone
```

```
pcs constraint order start neutron-netns-cleanup-clone then neutron-openvswitch-agent-clone
pcs constraint order start neutron-openvswitch-agent-clone then neutron-dhcp-agent-clone
pcs constraint order start neutron-dhcp-agent-clone then neutron-l3-agent-clone
pcs constraint order start neutron-l3-agent-clone then neutron-metadata-agent-clone
```

Colocation 约束主要用于根据资源 A 的节点位置来决定资源 B 的位置，即在启动资源 B 的时候，会依赖资源 A 的节点位置。例如将资源 A 与资源 B 进行 Colocation 约束，假设资源 A 已经运行在 Node1 上，则资源 B 也会在 Node1 上启动，而如果 Node1 故障，则资源 B 与 A 会同时切换到 Node2，而不是其中某个资源切换到 Node3。在 OpenStack 高可用集群配置中，通常需要将 Libvirt-d-compute 与 Neutron-openvswitch-agent 进行资源捆绑，要将 Nova-compute 与 Libvirt-d-compute 进行资源捆绑，则 Colocation 约束的配置如下：

```
pcs constraint colocation add nova-compute-clone with libvirt-d-compute-clone
pcs constraint colocation add libvirt-d-compute-clone with neutron-openvswitch-agent-compute-clone
```

Location 约束、Order 约束和 Colocation 约束是 Pacemaker 集群中最为重要的三个约束，通过这几个资源约束设置，集群中看起来彼此独立的资源就会按照预先设置有序运行，在 Openstack 高可用集群配置中，几乎全部资源的配置都会用到这几个约束。

3.7.3 集群资源类型

在 Pacemaker 集群中，各种功能服务通常被配置为集群资源，从而接受资源管理器的调度与控制，资源是集群管理的最小单位对象。在集群资源配置中，由于不同高可用模式的需求，资源通常被配置为不同的运行模式，例如 Active/Active 模式、Active/Passive 模式以及 Master/Master 模式和 Master/Slave 模式，而这些不同资源模式的配置均需要使用 Pacemaker 提供的高级资源类型，这些资源类型包括资源组、资源克隆和资源多状态等。

1. 资源组

在 Pacemaker 集群中，经常需要将多个资源作为一个资源组进行统一操作，例如将多个相关资源全部位于某个节点或者同时切换到另外的节点，并且要求这些资源按照一定的先后顺序启动，然后以相反的顺序停止，为了简化同时对多个资源进行配置，Pacemaker 提供了高级资源类型——资源组。通过资源组，用户便可并行配置多个资源，资源组的创建很简单，其语法格式如下：

```
pcs resource group add group_name resource_id [resource_id] ... [resource_id]
[--before resource_id | --after resource_id]
```

使用该命令创建资源组时，如果指定的资源组目前不存在，则此命令会新建一个资源组，如果指定的资源组已经存在，则此命令会将指定的资源添加到该资源组中。并且组中的资源会按照资源在该命令中出现的先后位置顺序启动，并以相反的顺序停止。在该命令

中, 还可使用 `--before` 和 `--after` 参数指定所添加的资源与组中已有资源的相对启动顺序。在为资源组添加资源时, 不仅可以将已有资源添加到组中, 还可以在创建资源的同时顺便将其添加到指定的资源组中, 命令语法如下:

```
pcs resource create resource_id standard:provider:type[type [resource_options] [op
operation_action operation_options] --group group_name
```

如下是资源组操作中经常使用的命令语法:

//将资源从组中删除, 如果该组中没有资源, 这个命令会将该组删除:

```
pcs resource group remove group_name resource_id...
```

//查看目前已经配置的资源组:

```
pcs resource group list
```

//创建名为 MyGroup 的资源组, 并添加资源 IPaddr 和 HAproxy:

```
pcs resource group add MyGroup IPaddr HAproxy
```

在 Pacemaker 集群中, 资源组所包含的资源数目是不受限的, 资源组中的资源具有如下的基本特性:

- ❑ 资源按照其指定的先后顺序启动, 如在前面示例的 MyGroup 资源组中, 首先启动 IPaddr, 然后启动 HAproxy。
- ❑ 资源按照其指定顺序的相反顺序停止, 如首先停止 HAproxy, 然后停止 IPaddr。
- ❑ 如果资源组中的某个资源无法在任何节点启动运行, 那么在该资源后指定的任何资源都将无法运行, 如 IPaddr 不能启动, 则 HAproxy 也不能启动。
- ❑ 资源组中后指定资源不影响前指定资源的运行, 如 HAproxy 不能运行, 但 IPaddr 却可以正常运行。

在集群资源配置过程中, 随着资源组成员的增加, 集群资源的配置工作将会明显减少, 因为管理员只需要添加资源到资源组中, 然后便可对资源组进行整体操作。资源组具有组属性, 并且资源组会继承组成员的部分属性, 主要被继承的资源属性包括 Priority、Target-role、Is-managed 等, 资源属性决定了资源在集群中的行为规范, 以及资源管理器可以对其进行哪些操作, 因此, 了解资源的常见属性也是非常必要的, 如下是资源属性中比较重要的几个属性解释及其默认值。

- ❑ Priority: 资源优先级, 其默认值是 0, 如果集群无法保证所有资源都处于运行状态, 则低优先权资源会被停止, 以便让高优先权资源保持运行状态。
- ❑ Target-role: 资源目标角色, 其默认值是 Started, 表示集群应该让这个资源处于何种状态, 允许值为:
 - Stopped: 表示强制资源停止;
 - Started: 表示允许资源启动, 但是在多状态资源的情况下不能将其提升为 Master 资源;
 - Master: 允许资源启动, 并在适当时将其提升为 Master。
- ❑ is-managed: 其默认值是 true, 表示是否允许集群启动和停止该资源, false 表示不

允许。

- ❑ **Resource-stickiness**: 默认值是 0, 表示该资源保留在原有位置节点的倾向程度值。
- ❑ **Requires**: 默认值为 `fencing`, 表示资源在什么条件下允许启动。

2. 资源克隆

克隆资源是 Pacemaker 集群中的高级资源类型之一, 通过资源克隆, 集群管理员可以将资源克隆到多个节点上并在启动时使其并行运行在这些节点上, 例如可以通过资源克隆的形式在集群中的多个节点上运行冗余 IP 资源实例, 并在多个处于 Active 状态的 IP 资源之间实现负载均衡。通常而言, 凡是其资源代理支持克隆功能的资源都可以实现资源克隆, 但需要注意的是, 只有已经规划为可以运行在 Active/Active 高可用模式的资源才能在集群中配置为克隆资源。配置克隆资源很简单, 通常在创建资源的过程中同时对其进行资源克隆, 克隆后的资源将会在集群中的全部节点上存在, 并且克隆后的资源会自动在其后添加名为 `clone` 的后缀并形成新的资源 ID, 资源创建并克隆资源的语法如下:

```
pcs resource create resource_id standard:provider:type|type [resource options]
--clone [meta clone_options]
```

克隆后的资源 ID 不再是语法中指定的 `Resource_id`, 而是 `Resource_id-clone`, 并且该资源会在集群全部节点中存在。在 Pacemaker 集群中, 资源组也可以被克隆, 但是资源组克隆不能由单一命令完成, 必须先创建资源组然后再对资源组进行克隆, 资源组克隆的命令语法如下:

```
pcs resource clone resource_id | group_name [clone_options]...
```

克隆后资源的名称为 `Resource_id-clone` 或 `Group_name-clone`。在资源克隆命令中, 可以指定资源克隆选项 (`clone_options`), 如下是常用的资源克隆选项及其意义。

- ❑ **Priority/Target - role/Is-manage**: 这三个克隆资源属性是从被克隆的资源中继承而来的, 具体意义可以参考上一节中的资源属性解释。
- ❑ **Clone-max**: 该选项值表示需要存在多少资源副本才能启动资源, 默认为该集群中的节点数。
- ❑ **Clone-node-max**: 表示在单一节点上能够启动多少个资源副本, 默认值为 1。
- ❑ **Notify**: 表示在停止或启动克隆资源副本时, 是否在开始操作前和操作完成后告知其他所有资源副本, 允许值为 `False` 和 `True`, 默认值为 `False`。
- ❑ **Globally-unique**: 表示是否允许每个克隆副本资源执行不同的功能, 允许值为 `False` 和 `True`。如果其值为 `False`, 则不管这些克隆副本资源运行在何处, 它们的行为都是完全相同的, 因此每个节点中有且仅有一个克隆副本资源处于 Active 状态。如果其值为 `True`, 则运行在某个节点上的多个资源副本实例或者不同节点上的多个副本实例完全不一样。如果 `Clone-node-max` 取值大于 1, 即一个节点上运行多个资源副本, 那么 `Globally-unique` 的默认值为 `True`, 否则为 `False`。

- **Ordered** : 表示是否顺序启动位于不同节点上的资源副本, True 为顺序启动, False 为并行启动, 默认值是 False。
- **Interleave** : 该属性值主要用于改变克隆资源或者 Masters 资源之间的 Ordering 约束行为, Interleave 可能的值为 True 和 False, 如果其值为 False, 则位于相同节点上的后一个克隆资源的启动或者停止操作需要等待前一个克隆资源启动或者停止完成才能进行, 而如果其值为 True, 则后一个克隆资源不用等待前一个克隆资源启动或者停止完成便可进行启动或者停止操作。Interleave 的默认值为 False。

在通常情况下, 克隆资源会在集群中的每个在线节点上都存在一个副本, 即资源副本数目与集群节点数目相等, 但是, 集群管理员可以通过资源克隆选项 Clone-max 将资源副本数目设为小于集群节点数目, 如果通过设置使得资源副本数目小于节点数目, 则需要通过资源位置约束 (Location Constraint) 将资源副本指定到相应的节点上, 设置克隆资源的位置约束与设置常规资源的位置约束类似。例如要将克隆资源 Web-clone 限制在 node1 节点上运行, 则命令语法如下:

```
pcs constraint location web-clone prefers node1
```

3. 资源多态

多状态资源是 Pacemaker 集群中实现资源 Master/Master 或 Master/Slave 高可用模式的机制, 并且多态资源是一种特殊的克隆资源, 多状态资源机制允许资源实例在同一时刻仅处于 Master 状态或者 Slave 状态。多状态资源的创建只需在普通资源创建的过程中指定 --Master 参数即可, Master/Slave 多状态类型资源的创建命令语法如下:

```
pcs resource create resource_id standard:provider:typetype [resource options]
--master [meta master_options]
```

多状态资源是一种特殊的克隆资源, 默认情况下, 多状态资源创建后也会在集群的全部节点中存在, 多状态资源创建后在集群中的资源名称形如 Resource_id-master。需要指出的是, 在 Master/Slave 高可用模式下, 尽管在集群中仅有一个节点上的资源会处于 Master 状态, 其他节点上均为 Slave 状态, 但是全部节点上的资源在启动之初均为 Slave 状态, 之后资源管理器会选择将某个节点的资源提升为 Master。另外, 用户还可以将已经存在的资源或资源组创建为多状态资源, 命令语法如下:

```
pcs resource master master/slave_name resource_id|group_name [master_options]
```

在多状态资源的创建过程中, 可以通过 Master 选项 (Master_options) 来设置多状态资源的属性, Master_options 主要有以下两种属性值:

- **Master-max** : 其值表示可将多少个资源副本由 Slave 状态提升至 Master 状态, 默认值为 1, 即仅有一个 Master。
- **Master-node-max** : 其值表示在同一节点中可将多少资源副本提升至 Master 状态, 默认值为 1。

在通常情况下，多状态资源默认会在每个在线的集群节点中分配一个资源副本，如果希望资源副本数目少于节点数目，则可通过资源的 Location 约束指定运行资源副本的集群节点，多状态资源的 Location 约束在实现的命令语法上与常规资源没有任何不同。此外，在配置多状态资源的 Ordering 约束时，可以指定对资源进行的操作是提升（Promote）还是降级（Demote）操作：

```
pcs constraint order [action] resource_id then [action] resource_id [options]
```

Promote 操作即是将对应的资源（resource_id）提升为 Master 状态，Demote 操作即是资源（resource_id）降级为 Slave 状态，通过 Ordering 约束即可设定被提升或降级资源的顺序。

3.7.4 集群资源规则

资源规则（Rule）使得 Pacemaker 集群资源具备了更强的动态调节能力，资源规则最常见的使用方式就是在集群资源运行时设置一个合理的粘性值（Resource-stickness），以防止资源回切到资源创建之初指定的高优先级节点上，即动态改变资源粘性值以防止资源意外回切。在大规模的集群资源配置中，资源规则的另一重要作用就是通过设置节点属性，将多个具有某一相同属性值的物理节点聚合到一个逻辑组中，然后通过资源的 Location 约束，利用节点组的这个共有节点属性值，将资源限制在该节点组上运行，即只允许此节点组中的节点运行该资源，在 OpenStack 高可用集群配置中，将会使用这种方式来限制不同的资源运行在不同的节点组上（控制节点组和计算节点组），大致的配置方式就是先为选定的节点设置某一自定义属性，以将其归纳到一个节点组，如下配置命令将计算节点和控制节点分别设置为不同的节点属性：

```
pcs property set --node compute1 osprole=compute
pcs property set --node compute2 osprole=compute
pcs property set --node controller1-vm osprole=controller
pcs property set --node controller2-vm osprole=controller
pcs property set --node controller3-vm osprole=controller
```

此处通过为节点分别设置不同的 osprole 属性值，将节点划分为两个集合，即计算节点组和控制节点组，将节点 compute1 和 compute2 归纳到 compute 节点组，节点 controller1-vm、controller2-vm 以及 controller3-vm 归纳到 controller 节点组，然后通过资源的 Location 约束将资源限制到不同的节点组中运行，配置命令如下：

```
pcs constraint location nova-compute-clone rule resource-discovery=exclusive
score=0 osprole eq compute
pcs constraint location nova-api-clone rule resource-discovery=exclusive score=0
osprole eq controller
```

在上述命令中，当 Rule 表达式“osprole=compute”或者“osprole=controller”成立，即 Rule 为 True，则执行对应资源的 Location 约束。此处，通过资源 Location 约束的

“resource-discovery=exclusive”配置，资源 nova-compute-clone 只能运行在 compute 节点组中，而 compute 组中只有 compute1 和 compute2 节点，因此 nova-compute-clone 只能在 compute1 和 compute2 上运行，绝不会在 controller1-vm、controller2-vm 及 controller3-vm 上运行。同样，nova-api-clone 资源只会出现在 controller 组中的三个节点上运行。绝不会在 compute1 和 compute2 节点上运行。

在 Pacemaker 集群中，每个资源的 Rule 都会包含一个或多个数字、时间及日期表达式，Rule 最终的取值则取决于多个表达式布尔运算的结果。布尔运算可以是管理员指定的逻辑与或者逻辑或操作，此外，Rule 的效果总是以 Constraint 的形式体现。因此，Rule 通常在 Constraint 命令中配置，如下语句是配置资源 Rule 的语法格式：

```
pcs constraint rule add constraint_id [rule_type] [score=score ] [id=rule_id]
expression|date_expression|date_spec options
```

如果忽略 Score 值，则使用默认值 INFINITY，如果忽略 ID，则自动从 Constraint_id 生成一个规则 ID，而 Rule_type 可以是字符表达式或者日期表达式。需要注意的是，在删除资源 Rule 时候，如果此 Rule 是 Constraint 中的最后一个 Rule，则该 Constraint 将被删除，删除资源 Rule 语法如下：

```
pcs constraint rule remove rule_id
```

资源 Rule 的表达式主要分为节点属性表达式和时间 / 日期表达式，节点属性表达式由以下几个部分组成。

- Value：用户提供的用于同节点属性值进行比较的值。
- Attribute：节点属性变量名，其值即是 Value 要匹配的节点属性值。
- Type：确定使用哪种类型的值匹配，允许的值包括字符串、整数、版本号 (Version)。
- Operation：操作符，确定用户提供的 Value 与节点 Attribute 的值如何匹配，主要包括以下几种操作符。
 - lt：如果 Value 小于 Attribute 的值，表达式为正 True；
 - gt：如果 Value 大于 Attribute 的值，表达式为正 True；
 - lte：如果 Value 小于等于 Attribute 的值，表达式为正 True；
 - gte：如果 Value 大于等于 Attribute 的值，表达式为正 True；
 - eq：如果 Value 等于 Attribute 的值，表达式为正 True；
 - ne：如果 Value 不等于 Attribute 的值，表达式为正 True；
 - defined：如果表达式中的 Attribute 在节点中有定义，则表达式为 True；
 - not_defined：如果节点中没有定义表达式中的 Attribute，则表达式为 True。

要通过 Rule 的节点属性表达式来确定资源的 Location，则通常的命令语法如下：

```
pcs resource constraint location resource_id rule [rule_id] [role=master|slave]
[score=score expression]
```


此处的表达式可以是以下几种形式：

- ❑ `defined|not_defined attribute`
- ❑ `attribute lt|gt|lte|gte|eq|ne value`
- ❑ `date [start=start] [end=end] operation=gt|lt|in-range`
- ❑ `date-spec date_spec_options`

在 OpenStack 高可用集群配置中，使用最多的是第二种形式的表达式，例如要限制 Nova-compute 服务仅运行在计算节点上，则可以通过如下 Rule 和 Location 配置实现：

```
pcs constraint location nova-compute-clone rule\
resource-discovery=exclusive score=0 osprole eq compute
```

上述命令中，“osprole eq compute”即是 Rule 的表达式，其中 osprole 是节点的 Attribute，Compute 是用户指定的节点属性值，该表达式的操作符是等于符号（eq）。该命令语句中的规则表达式的意思就是，当节点的 osprole 值等于用户指定值（compute）的时候，则 Rule 表达式为 True（计算节点属性中已经预先设置了 osprole 属性值为 compute）。

3.8 本章小结

就目前的 OpenStack 高可用集群部署而言，Pacemaker 已然成为主流的 OpenStack 集群资源管理器。除了官方社区指定 Pacemaker 作为 OpenStack 高可用资源管理器外，各大 OpenStack 服务供应商的高可用解决方案几乎均选用 Pacemaker 作为集群资源管理器以提供高可用的 OpenStack 服务。例如 Redhat 的 Openstack Platform 系列发行版本、Mirantis 的 Fuel 系列以及 HPE 的 Helion 等解决方案均采用了 Pacemaker 集群资源管理器。作为 Linux 系统中最为流行的高可用资源管理器，Pacemaker 提供了丰富强大的集群功能以满足用户不同的高可用需求，本章不仅从内部结构和功能特性等方面对集群资源管理器 Pacemaker 进行了详细介绍，还结合 OpenStack 的高可用服务部署对 Pacemaker 在集群资源配置管理中的使用进行了列举，本章内容也是实战部署 OpenStack 高可用集群的基础，只有在充分理解和掌握本章内容的基础上，才能真正部署和运维 OpenStack 高可用集群。

集群负载均衡系统

负载均衡器 (Load Balancer, LB) 是一组能够将 IP 数据流以负载均衡形式转发到多台物理服务器的集成软件。在实际使用中,有硬件负载均衡器和软件负载均衡器之分,硬件负载均衡器主要是在访问网络和服务器之间配置物理负载均衡设备,客户端对物理服务器的访问请求首先会抵达负载均衡设备,然后再由负载均衡设备根据一定的负载算法转发到后端服务器。相比而言,软件负载均衡器不需要特定的物理设备,只需在相应的操作系统上部署具有负载均衡功能的软件即可,而且从功能配置和成本预算来看,软件负载均衡器要比硬件负载均衡器简单和实惠,并且随着开源技术的发展,软件负载均衡器在功能和稳定性方面也不比硬件产品差很多。因此,在 Redhat 和 Mirantis 等 OpenStack 领导厂商的高可用集群部署方案中,负载均衡系统几乎全采用软件实现。此外,很多大型网站的主流 Web 服务架构也均采用软件负载均衡器来实现分流访问和架构的可扩展性,可以说,软件负载均衡器在互联网行业的发展和长期实践中,其功能和稳定性都得到了充分的证明和有效使用。

从市场占有率来看,硬件负载均衡器的领导厂商毫无疑问是美国的 F5 公司,其 BIG-IP 系列负载均衡器在金融、制造和互联网等行业被普遍使用,而国内的深信服和天融信等厂商在负载均衡领域也具有一定的产品和知名度。相对软件负载均衡器而言,硬件负载均衡器虽然更具稳定性,但是其成本和后期维护使得很多企业难以接受。而在开源领域,软件负载均衡器被大量使用,其中最为知名的便是 HAProxy 和 Keepalived。OpenStack 官方社区推荐的高可用集群部署便是基于 HAProxy 和 Keepalived 的方案。Keepalived 主要使用 Linux 虚拟服务器 (Linux Virtual Server, LVS) 进行负载均衡和故障切换 (Failover),而 HAProxy 主要进行负载均衡和实现基于 TCP 及 HTTP 协议的服务高可用,为了实现更具稳定性和扩展性的高可用环境,用户可以将 HAProxy 和 Keepalived 两个软件进行组合使用,

在利用 HAProxy 对基于 HTTP 和 TCP 服务的快速负载均衡和可扩展性等优势的同时,借助 Keepalived 来实现服务的故障切换,通过两个软件的组合使用,用户不仅实现了负载均衡和服务高可用,同时还实现了故障路由的自动切换并保证了服务的可持续性。

在 Openstack 高可用集群部署中,服务的负载均衡和高可用主要有两种主流的实现方案,即 HAProxy+Keepalived 和 Pacemaker+HAProxy 方案。由于 OpenStack 服务组件多样,不同服务均需要进行特定的高可用设计,并且从集群资源统一调度和集群稳定性的角度考虑,后一种方案是多数 OpenStack 厂商的高可用部署方案首选,但是选用后一方案并不意味着 Keepalived 在 OpenStack 高可用集群部署中不被使用。由于 Keepalived 的主要作用之一是进行虚拟路由的故障切换,其在 Neutron 的 L3 高可用设计与实现中起着举足轻重的作用。因此,为了提供后续 OpenStack 高可用集群部署的负载均衡系统理论基础,本章对 Keepalived 和 HAProxy 的概念和基本配置进行了讲解,而不论选用哪种 Openstack 高可用部署方案,本章内容都是必须掌握的基础。

4.1 Keepalived 概述与配置

4.1.1 Keepalived 及 LVS 概述

Keepalived 最初是由 Alexandre Cassen 使用 C 语言编写的开源软件项目,其项目实现的主要目标是简化 LVS 项目的配置并增强其稳定性,即 Keepalived 是对 LVS 项目的扩展增强。Keepalived 为 Linux 系统和基于 Linux 的架构提供了负载均衡和高可用能力,其负载均衡功能主要源自集成在 Linux 内核中的 LVS 项目模块 IPVS (IP Virtual Server),基于 IPVS 提供的 4 层 TCP/IP 协议负载均衡,Keepalived 也具备负载均衡的功能,此外,Keepalived 还实现了基于多层 TCP/IP 协议 (3 层、4 层、5/7 层) 的健康检查机制,因此,Keepalived 在 LVS 负载均衡功能的基础上,还提供了 LVS 集群物理服务器池健康检查和故障节点隔离的功能。除了扩展 LVS 的负载均衡服务器健康检查能力,Keepalived 还基于虚拟路由冗余协议 (Virtual Route Redundancy Protocol, VRRP) 实现了 LVS 负载均衡服务器的故障切换转移,即 Keepalived 还实现了 LVS 负载均衡器的高可用性。根据 Keepalived 官方网站^①的陈述,Keepalived 就是为 LVS 集群节点提供健康检查和为 LVS 负载均衡服务器提供故障切换的用户空间进程。

图 4-1 为 Keepalived 的原理架构图,从图中可以看到,Keepalived 的多数核心功能模块均位于用户空间,而仅有 IPVS 和 NETLINK 模块位于内核空间,但是这两个内核模块正是 Keepalived 实现负载均衡和路由高可用的核心模块,其中的 NETLINK 主要用于提供高级路由及其相关的网络功能。Keepalived 的大部分功能模块位于用户空间,其中几个核心功能模块的介绍如下。

① <http://www.keepalived.org>

- ❑ WatchDog：其主要负责监控 Checkers 和 VRRP 子进程的运行状况。
- ❑ Checkers：此功能模块主要负责真实服务器的健康检查（HealthChecking），是 Keepalived 最主要的功能之一，因为 HealthChecking 是负载均衡功能稳定运行的基础，LVS 集群节点的故障隔离和重新加入均依赖于 HealthChecking 的结果。
- ❑ VRRP Stack：此功能模块主要负责负载均衡器之间的故障切换，如果集群架构中仅使用一个 LVS 负载均衡器，由于本身不具备故障切换的条件，则 VRRP Stack 不是必须的。
- ❑ IPVS Wrapper：此模块主要用来发送设定的规则到内核 IPVS 代码。Keepalived 的设计目标是构建高可用的 LVS 负载均衡群集，Keepalived 在运行中将会通过 IPVS Wrapper 模块调用 IPVSAdmin 工具来创建虚拟服务器，检查和管理 LVS 集群物理服务器池。
- ❑ Netlink Reflector：此功能模块主要用来设定 VRRP 的 VIP 地址并提供相关的网络功能，该模块通过与内核中的 NETLINK 模块交互，从而为 Keepalived 提供路由高可用功能。

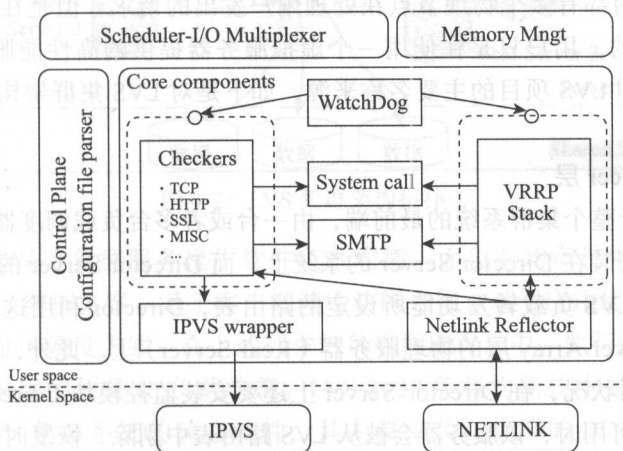


图 4-1 Keepalived 原理架构图

从 Keepalived 的实现原理和功能来看，Keepalived 是开源负载均衡项目 LVS 的增强和虚拟路由协议 VRRP 实现的集合，即 Keepalived 通过整合和增强 LVS 与 VRRP 来提供高可用的负载均衡系统架构，为了更好地掌握 Keepalived，本节在对 Keepalived 介绍的同时，对 LVS 和 VRRP 也将进行简要介绍。

1. Linux 虚拟服务器——LVS

LVS 是 Linux Virtual Server 的简称，即 Linux 虚拟服务器，最初是由国内的章文嵩博士于 1998 年发起的自由软件项目，它的官方网址是 www.linuxvirtualserver.org（对应的中文网站为 <http://zh.linuxvirtualserver.org/>）。目前 LVS 项目已经被集成到 Linux 内核中，

在 Linux2.4 内核之前, 要使用 LVS 功能就必须重新编译内核以支持 LVS 功能模块, 但从 Linux2.4 之后, LVS 的各个功能模块已经被内置入 Linux 内核中, 无需进行额外的安装和内核重新编译, 便可直接使用 LVS 提供的各种功能。LVS 具有良好的可靠性、可扩展性和可操作性, 加上其实现最优的集群服务性能所需的低廉成本, LVS 的负载均衡功能经常被用于高性能、高可用的服务器群集中。

从 1998 年至今, LVS 已经发展成为非常成熟的开源项目, 基于 LVS 技术可以实现很多高伸缩、高可用的网络服务, 例如 Web 服务、Cache 服务、DNS 服务、FTP 服务、MAIL 服务、视频/音频点播服务等, LVS 的功能也在发展过程中得到了很多用户的实践验证, 例如很多著名网站和组织都在使用基于 LVS 架构的集群系统, 包括 Linux 门户网站 (www.linux.com)、向 RealPlayer 提供音频视频服务的 Real 公司 (www.real.com)、全球最大的开源网站 (sourceforge.net) 等都是 LVS 项目的使用者。

在基于 LVS 项目架构的服务器集群系统中, 通常包含三个功能层次: 最前端的负载均衡层 (Load Balancer)、中间的物理服务器群组层 (Server Array) 以及最底端的数据共享存储层 (Shared Storage), 典型的 LVS 集群架构如图 4-2 所示。在 LVS 负载均衡集群架构中, 尽管整个集群内部有多个物理节点在处理用户发出的请求, 但是在用户看来, 所有的内部应用都是透明的, 用户只是在使用一个虚拟服务器提供的高性能服务, 这也是 Linux 虚拟服务器项目, 即 LVS 项目的主要名称来源, 如下是对 LVS 集群架构中各个层次的功能描述。

(1) Load Balancer 层

负载均衡层位于整个集群系统的最前端, 由一台或者多台负载调度器 (Director Server) 组成, LVS 模块就安装在 Director Server 的系统上, 而 Director Server 的主要功能类似路由器, 其包含了完成 LVS 负载转发功能所设定的路由表, Director 利用这些路由表信息把用户的请求分发到 Server Array 层的物理服务器 (Real Server) 上。此外, 为了监测各个 Real Server 服务器的健康状况, 在 Director Server 上还要安装监控模块 Ldirectord, 而当监控到某个 Real Server 不可用时, 该服务器会被从 LVS 路由表中剔除, 恢复时又会重新加入。

(2) Server Array 层

服务器阵列或服务器池由一组实际运行应用服务的物理机器组成, Real Server 可以是 Web 服务器、Mail 服务器、FTP 服务器、DNS 服务器以及视频服务器中的一个或者多个的组合。每个 Real Server 之间通过高速的 LAN 或分布各地的 WAN 相连接。在实际应用中, 为了减少资源浪费, Director Server 也可以同时兼任 Real Server 的角色, 即在 Real Server 上同时部署 LVS 模块。

(3) Shared Storage 层

存储层是为所有 Real Server 提供共享存储空间和内容一致性的存储区域, 在物理实现上, 该层一般由磁盘阵列设备组成。而为了提供一致性的内容, 通常利用 NFS 网络文件系统提供集群的共享数据, 但是 NFS 在繁忙的业务系统中, 性能并不是很好, 此时可以采用

集群文件系统,例如 Red hat 的 GFS 文件系统和 IBM 的 GPFS 文件系统等。

在 LVS 集群架构中, Director Server 是整个 LVS 集群的核心,目前,用于 Director Server 的操作系统仍然局限于 Linux 和 FreeBSD, Linux2.4 以后的内核不用任何设置就可以支持 LVS 功能,而 FreeBSD 作为 Director Server 的应用目前来看不是很多,性能也不如 Linux 理想。但是对于 Real Server 而言,其操作系统几乎可以是所有的系统平台,如 Linux、Windows、Solaris、AIX、BSD 系列都能得到很好的支持。

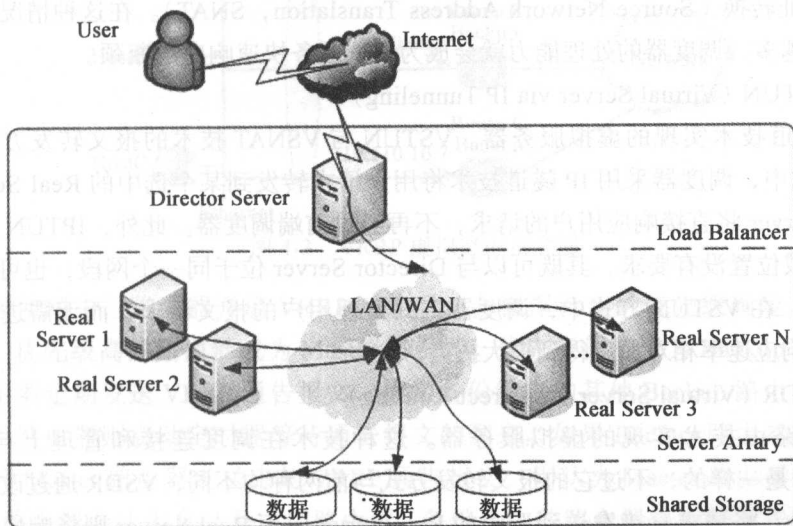


图 4-2 LVS 集群架构拓扑

LVS 的核心功能是为集群服务提供软件负载均衡,而负载均衡技术有很多实现方案,如基于 DNS 域名轮流解析方案、基于客户端调度访问方案、基于应用层系统负载的调度方案,以及基于 IP 地址的调度方案。在上述列举的负载调度算法中,执行效率最高的是 IP 负载均衡技术,LVS 采用的便是 IP 负载均衡。LVS 的 IP 负载均衡技术是通过 IPVS 模块来实现的,IPVS 是 LVS 集群系统的核心软件,其主要安装在集群的 Director Server 上,并在 Director Server 上虚拟出一个服务 IP 地址,用户对服务的访问只能通过该虚拟 IP 地址实现。这个虚拟 IP 通常称为 LVS 的 VIP (Virtual IP),用户的访问请求首先经过 VIP 到达负载调度器,然后由负载调度器从 Real Server 列表中按照一定的负载均衡算法选取一个服务节点响应用户的请求。在这个过程中,当用户的请求到达 Director Server 后,Director Server 如何将请求转发到提供服务的 Real Server 节点,而 Real Server 节点又如何将数据返回给用户,这是 IPVS 实现负载均衡的核心技术。IPVS 实现数据路由转发的机制有三种,分别是 NAT、TUN 和 DR 技术,三种实现方法的描述如下。

(1) VSNAT (Virtual Server via Network Address Translation)

即通过网络地址转换的虚拟服务器技术。在这种负载转发方案中,当用户的请求到达调度器时,调度器自动将请求报文的目标 IP 地址 (VIP) 替换成 LVS 选中的后端 Real

Server 地址, 同时报文的目标端口也替换为选中的 Real Server 对应端口, 最后将报文请求发送给选中的 Real Server 进行处理。当 Real Server 处理完请求并将结果数据返回给用户时, 需要再次经过负载调度器, 此时调度器进行相反的地址替换操作, 即将报文的源地址和源端口改成 VIP 地址和相应端口, 然后把数据发送给用户, 完成整个负载调度过程。

可以看出, 在这种方式下, 用户请求和响应报文都必须经过 Director Server 进行地址转换, 请求时进行目的地址转换 (Destination Network Address Translation, DNAT), 响应时进行源地址转换 (Source Network Address Translation, SNAT)。在这种情况下, 如果用户请求越来越多, 调度器的处理能力就会成为集群服务快速响应的瓶颈。

(2) VSTUN (Virtual Server via IP Tunneling)

即 IP 隧道技术实现的虚拟服务器。VSTUN 与 VSNAT 技术的报文转发方法不同, 在 VSTUN 方式中, 调度器采用 IP 隧道技术将用户请求转发到某个选中的 Real Server 上, 而这个 Real Server 将直接响应用户的请求, 不再经过前端调度器。此外, IPTUN 技术对 Real Server 的地域位置没有要求, 其既可以与 Director Server 位于同一个网段, 也可位于独立网络中。因此, 在 VSTUN 方式中, 调度器将只处理用户的报文请求, 而无需进行转发, 故集群系统的响应速率相对而言得到极大提高。

(3) VSDR (Virtual Server via Direct Routing)

即直接路由技术实现的虚拟服务器。这种技术在调度连接和管理上与 VSNAT 和 VSTUN 技术是一样的, 不过它的报文转发方式与前两种均不同, VSDR 通过改写请求报文的 MAC 地址, 将请求直接发送到选中的 Real Server, 而 Real Server 则将响应直接返回给客户端。因此, 这种技术不仅避免了 VSNAT 中的 IP 地址转换, 同时也避免了 VSTUN 中的 IP 隧道开销, 所以 VSDR 是三种负载调度机制中性能最高的实现方案。但是, 在这种方案下, Director Server 与 Real Server 必须在同一物理网段上存在互联。

2. 虚拟路由冗余协议——VRRP

虚拟路由冗余协议 (Virtual Router Redundancy Protocol, VRRP) 是由因特网工程任务组 (Internet Engineering Task Force, IETF) 提出的解决局域网中配置静态网关出现单点失效现象的路由协议。VRRP 是一种容错协议, 其主要目的是解决路由单点故障的问题。VRRP 协议将局域网内的一组路由器虚拟为单个路由, 通常将其称为一个路由备份组, 而这组路由器内包括一个 Master 路由 (即活动路由器) 和若干个 Backup 路由 (即备份路由器), VRRP 虚拟路由示意图如图 4-3 所示。在图 4-3 中, RouterA、RouterB 和 RouterC 属于同一个 VRRP 组, 组成一个虚拟路由器, 而由 VRRP 协议虚拟出来的路由器拥有自己的 IP 地址 10.110.10.1, 而备份组内的路由器也有自己的 IP 地址 (如 Master 的 IP 地址为 10.110.10.5, Backup 的 IP 地址为 10.110.10.6 和 10.110.10.7)。

在实际使用中, 局域网内的主机仅仅知道这个虚拟路由器的 IP 地址 10.110.10.1, 而并不知道具体的 Master 路由器的 IP 地址以及 Backup 路由器的 IP 地址。局域网内的主机将自己的默认路由下一跳地址设置为该虚拟路由器的 IP 地址 10.110.10.1, 之后, 网络内的主机

就通过这个虚拟的路由器来与其他网络进行通信。在通信过程中,如果备份组内的 Master 路由器故障,则 Backup 路由器将会通过选举机制重新选出一个新的 Master 路由器,从而继续向网络内的主机提供路由服务,最终实现了路由功能的高可用。

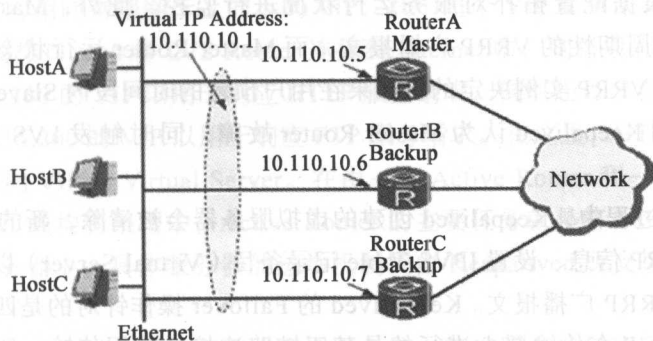


图 4-3 VRRP 虚拟路由示意图

在实际使用中,路由器开启 VRRP 功能后,会根据设定的优先级确定自己在备份组中的角色。优先级高的路由器成为 Master 路由器,优先级低的成为 Backup 路由器,并且 Master 路由器定期发送 VRRP 通告报文,通知备份组内的其他 Backup 路由器自己工作正常,而备用路由器则启动定时器等待通告报文的到来。如果 Backup 路由器的定时器超时后仍未收到 Master 路由器发送来的 VRRP 通告报文,则认为 Master 路由器已经故障,此时 Backup 路由器会认为自己是主用路由器(备份组内的路由器会根据优先级选举出新的 Master 路由器),并对外发送 VRRP 通告报文。此外,VRRP 在提高路由可靠性的同时,还简化了主机的路由配置,在具有多播或广播能力的局域网中,借助 VRRP 能在某台路由器出现故障时仍然提供高可靠的默认链路,有效避免单一链路发生故障后网络中断的问题,并且无需修改主机动态路由协议、路由发现协议等配置信息。

4.1.2 Keepalived 工作原理

从 4.1.1 节对 Keepalived 的介绍可知,Keepalived 本质上是提供数据流转发与服务器健康检查并具备故障切换的高可用路由,而数据转发与健康检查是对 LVS 功能的扩展和增强,因此也可以认为 Keepalived 是运行在用户空间的 LVS 路由(LVS Router)进程。在实际应用中,Keepalived 通常部署在两台主备或一主多备的服务器上,即 Keepalived 进程既运行在 Active/Master 状态的 LVS Router 中,也运行在 Passive/Slave 状态的 LVS Router 中,而所有运行 Keepalived 进程的 LVS Router 都遵循虚拟路由冗余协议 VRRP。在 VRRP 的协议框架下,作为 Master 的 Router 将会处理两个主要任务,即转发客户端访问请求到后端物理服务器以进行负载均衡和周期性的发送 VRRP 协议报文,而作为 Slave 的 Routers 则负责接收 VRRP 报文,如果某一时刻作为 Slave 的 Routers 接收 VRRP 报文失败,则认为 Master Router 故障,并从 Slave Routers 中重新选举产生一个新的 Master Router。

Keepalived 是一个与 LVS Router 相关的控制进程，在 RHEL7/Centos7 系统中，Keepalived 由 Systemctl 命令通过读取 /etc/keepalived/keepalived.conf 配置文件来启动。在遵循 VRRP 协议的 Master Router 中，Keepalived 进程会启动内核中的 LVS 服务以创建虚拟服务器，并根据配置拓扑对服务运行状况进行监控。此外，Master Router 还会向 Slave Routers 发送周期性的 VRRP 广播报文，而 Master Router 运行状态的正常与否是由 Slave Routers 上的 VRRP 实例决定的。如果在用户预置的时间段内 Slave Router 不能接收到 VRRP 报文，则 Keepalived 认为 Master Router 故障，同时触发 LVS Router 的 Failover 操作。

在 Failover 的过程中，Keepalived 创建的虚拟服务器会被清除，新的 Master Router 将接管 VIP、发送 ARP 信息、设置 IPVS Table 记录条目（Virtual Server）以及物理服务器的健康检查和发送 VRRP 广播报文。Keepalived 的 Failover 操作针对的是四层 TCP/IP 协议，即传输层，因为 TCP 在传输层上进行的是基于链路连接的数据传输。所以，当服务器在响应 TCP 请求时，如果出现设置时间段的 Timeout，则 Keepalived 的健康检查机制将会监测到该情况并认为该服务器故障，然后将其从服务器池中移除（故障服务器隔离）。

图 4-4 是基于 Keepalived 设计的具有二层拓扑的负载均衡架构，该架构分为两个层次。第一层为负载均衡层，由一个 Active 和多个 Backup 的 LVS Routers 组成，其中，每个 LVS Router 都配置有两个网络接口，一个接入 Internet 网络，另一个接入内部私有网络，Active 的 LVS Router 在这两个网络接口间进行数据转发。在图 4-4 的负载均衡架构中，位于第一层的 LVS Routers 和第二层的物理服务器通过私网接口接入相同的局域网中，Active 的 LVS Router 通过 NAT 技术将 Internet 数据流转发到私网物理服务器上，而这些位于第二层的物理服务器运行着最终响应请求的服务。

位于二层私网中的服务器在与 Internet 交互时必须经过主 LVS Router 的 NAT 转发，并且对于外部网络中的客户端而言，访问二层私网中的物理服务器就如访问同处 Internet 网络中的服务，因为从客户端的角度来看，访问请求的目的地址正是位于主 LVS Router 上的 VIP 地址，而该 VIP 与客户端地址处于相同网络中，VIP 还可以是管理员指定的互联网域名，如 www.example.com。VIP 在 Keepalived 的配置中通常被指定到一个或者多个虚拟服务器上，而虚拟服务器的主要任务便是监听 VIP 及相应端口上的请求，当主 LVS Router 进行 Failover 操作的时候，VIP 会从一个 LVS Router 转移到另一个 LVS Router（因此 VIP 也称为浮动 IP），从而保证

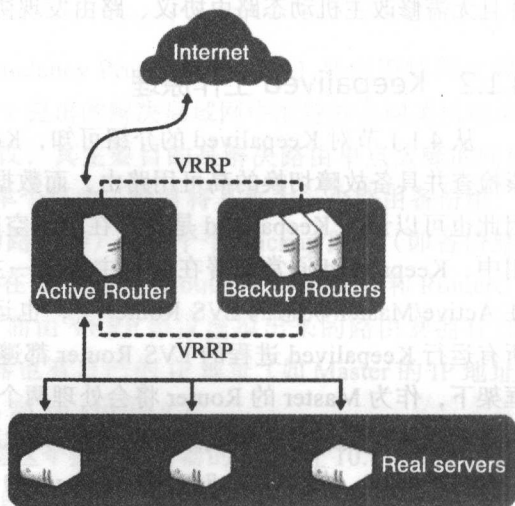


图 4-4 基于 Keepalived 的两层拓扑负载均衡架构

了对外提供服务的 VIP 具有高可用性。

在 Keepalived 负载均衡架构的 VIP 配置中, 每个将 LVS Router 连接到 Internet 的物理网卡接口均可配置多个 VIP, 并且每个 VIP 对应着不同的 Virtual Server, 即多个 Virtual Servers 可以同时监听相同物理网卡上的不同 VIP, 其中每个 VIP 都对应着不同的服务。例如, Linux 系统中的接口 eth0 将 LVS Router 连接到 Internet 中, 则可以在 eth0 上配置一个地址为 192.168.115.100 的 VIP 以用于响应 HTTP 服务请求, 同时还可以在 eth0 上配置另一个地址为 192.168.115.200 的 VIP 以用于响应 FTP 服务请求。在这里, HTTP 服务和 FTP 服务均对应着监听不同 VIP 的 Virtual Server。在由一个 Active Router 和一个 Backup Router 组成的 Keepalived 负载均衡架构中, Active Router 的主要任务就是将 VIP 上的请求转发到选中的某个后端服务器上, 具体服务器的选举机制则由 Keepalived 所支持的负载均衡算法来决定。

此外, Active Router 还负责动态监控后端服务器上特定服务的健康状况, 监控方式主要是 Keepalived 自带的三种健康检测机制, 即简单 TCP 连接、HTTP 和 HTTPS。就简单 TCP 连接检测方式, Active Router 会周期性地对服务器上某个特定端口进行 TCP 连接, 如果 TCP 连接超时或者中断则认为服务不可用, 而对于 HTTP 和 HTTPS 检测方式, Active Router 通过周期性地抓取 (Fetch) 请求 URL 并验证其内容来判断服务的可用性。与此同时, Backup Router 一直处于 Standby 状态, LVS Router 的 Failover 由 VRRP 来处理。在 Keepalived 进程启动的时候, 所有 LVS Routers 会加入一个用来接收和发送 VRRP 广播的多播组, 由于 VRRP 是一种基于优先级的协议, 因此在启动之初优先级高的 LVS Router 会被选举为 Master Router, 而 Master Router 将会周期性地向多播组中的成员发送 VRRP 广播。如果多播组中的 Backup Routers 在一定时间内接收 VRRP 广播失败, 则重新选举新的 Master Router, 新的 Master Router 将会接管 VIP 并广播地址解析协议 (Address Resolution Protocol, ARP) 信息。而当故障 Router 重新恢复后, 根据该 Router 的优先级情况, 其可能恢复到 Master 状态也可能保持为 Backup 状态。

图 4-4 中的两层负载均衡架构是最常见的部署环境, 主要用于很多数据源变化不是很频繁的数据请求服务中, 如静态 Web 页面站点, 因为后端独立服务器 (Real Servers) 之间不会自动进行数据同步。图 4-5 为基于 Keepalived 的三层负载均衡架构, 在三层负载均衡架构中, 前端的 LVS Router 负责将访问请求转发到物理服务器 (Real Servers) 中, 然后 Real Server 再通过网络形式访问可共享的数据源。对于数据请求比较繁忙的 FTP 站点, 三层架构是最为理想的负载均衡架构, 在这种架构下, 可供访问的数据源集中存储在高可用的集群服务器上, Real Servers 通过 NFS 共享目录或者 Samba 文件共享等网络文件系统形式来访问数据。此外, 类似的三层负载均衡架构在需要提供中心化及数据库事务处理高可用的 Web 站点中也被普遍使用, 如果将 Keepalived 负载均衡器配置为 Active/Active 双活模式, 则还可以将三层负载均衡架构同时用于提供 FTP 和 Web 数据库服务。

4.1.3 Keepalived 调度算法

Keepalived 支持多种负载均衡调度算法, 因此 Keepalived 可以采取灵活多变的方式将请求负载转发到后端服务器池中。相对于很多缺少灵活性的调度算法, 负载均衡是更为优越的选择, 如基于 Round-Robin DNS 的调度算法, 其 DNS 固有的层次结构和客户端缓存机制使得很多时候不能实现真正的负载均衡。此外, LVS Router 所采用的底层过滤机制相对于上层的应用请求转发更具优势, 因为基于网络数据包层面的负载均衡仅需极少的计算资源并具有随意的可扩展性。

Keepalived 所使用的负载均衡调度机制由集成到内核中的 IPVS 模块提供, IPVS 是 LVS 项目的核心功能模块, 其设计的主要目的之一就是解决单 IP 多服务器的工作环境, IPVS 模块使得基于 TCP/IP 传输层 (第 4 层) 的数据交换成为可能。在实际使用中, IPVS 会在内核中创建一个名为 IPVS Table 的表, 该表记录了后端服务器的地址及服务运行状态, 通过 IPVS Table, Keepalived 便可跟踪并将请求路由到后端物理服务器中, 即 LVS Router 利用此表将来自 Keepalived 虚拟服务器地址的请求转发到后端服务器池中, 同时将后端服务器的处理结果转发给客户端。此外, IPVS Table 的表结构主要取决于管理员对指定的虚拟服务器所设置的负载均衡算法, Keepalived 支持以下几种负载均衡算法。

(1) Round-Robin

即所谓的轮询负载均衡, 在这种算法中, 服务请求会被依次转发到服务器池中的每一个服务器上, 而不去评估服务器的当前负载或者处理能力, 服务器池中的每一个服务器都被平等对待。如果使用 Round-Robin 负载均衡算法, 每台后端服务器会轮询依次处理服务请求。

(2) Weighted Round-Robin

即加权 Round-Robin 算法, 是对 Round-Robin 算法的一种扩展。在这种算法中, 请求被依次转发到每一台服务器上, 但是当前负载较轻或者计算能力较大的服务器会被转发更多的请求, 服务器的处理能力通过用户指定的权重因子来决定, 权重因子可以根据负载信息动态上调或者下调。如果服务器的配置差别较大, 导致不同服务器的处理能力相差较大, 则加权的 Round-Robin 算法会是不错的选择, 但是如果请求负载频繁变动, 则权重较大的

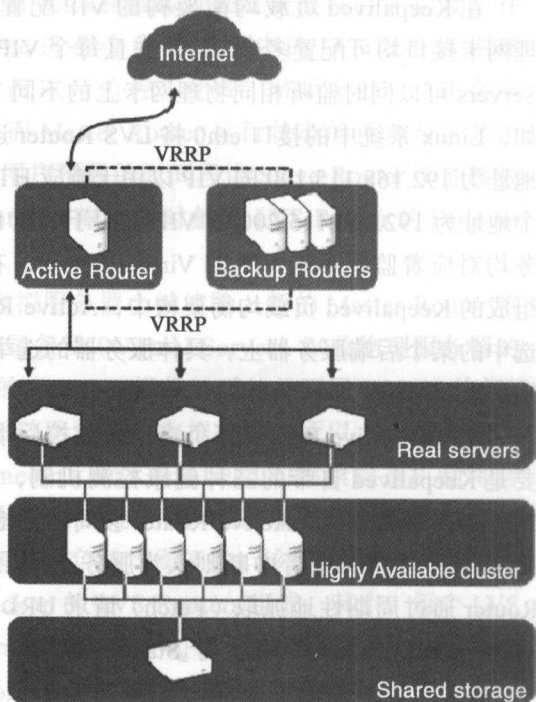


图 4-5 基于 Keepalived 的三层拓扑负载均衡架构

服务器可能会超负荷工作。

(3) Least-Connection

即最少连接算法,在这种算法中,请求被转发到活动连接较少的服务器上。在 Keepalived 的实际使用中, LVS Router 一直在利用内核中的 IPVS Table 来记录后端服务器的活动连接,从而动态跟踪每个服务器的活动连接数。最少连接数算法是一种动态决策算法,它比较适合服务器池中每个成员的处理能力都大致相当,同时负载请求又频繁变化的场景,如果不同服务器有不同的处理能力,则下面的加权最少连接数算法较为合适。

(4) Weighted Least-Connections

即加权最少连接数算法,在这种算法中,路由会根据服务器的权重,转发更多的请求到连接数较少的服务器上。服务器的处理能力通过用户指定的权重因子来决定,权重因子可以根据负载信息动态上调或者下调。一般来说,服务器加权算法主要用于集群存在不同类型服务器,而服务器配置和处理能力相差较大的场景中。

(5) Destination Hash Scheduling

即目标地址哈希算法,通过在静态 Hash 表中查询目的 IP 地址来确定请求要转发的服务器,这类算法主要用于缓存代理服务器集群中。

(6) Source Hash Scheduling

即源地址哈希算法,通过在静态 Hash 表中查询源 IP 地址来确定请求要转发的服务器,这类算法主要应用于存在多防火墙的 LVS Router 中。

(7) Shortest Expected Delay

即最小延时算法,在这种算法中,请求被转发到具有最小连接响应延时的服务器上。

4.1.4 Keepalived 路由方式

Keepalived 将请求转发到服务器池中的某台具体服务器依赖的是 4.1.3 节中介绍的调度算法,而在调度算法从服务器池中选中某台具体服务器来处理本次请求之后, LVS Router 又如何将接收到的请求转发给该服务器,这便是 Keepalived 路由方案所要解决的问题。4.1.1 节经介绍了 LVS 项目中的 IPVS 模块对数据进行路由转发的三种方式,即 NAT、TUN 和 DR 方式。Keepalived 负载均衡器将客户端请求转发到后端服务器仍然是基于内核中的 IPVS 模块,即 Keepalived 也使用这三种路由转发方式进行客户端请求转发,但在实际使用中,考虑到硬件的多样性和集成负载均衡器到现有网络的灵活性,通常采用的 Keepalived 路由转发机制为 NAT 和 DR 两种方式,而 DR 路由一般为 Keepalived 的首选方案。

1. NAT 路由

图 4-6 为基于 NAT 路由实现的 Keepalived 负载均衡器,在 NAT 机制下,每个 LVS Router 需要两个网络接口。假设 eth0 为接入 Internet 的网络接口,则 eth0 上配置有一个真实的 IP 地址,同时还配置了一个浮动 IP 地址 (Floating IP); 假设 eth1 为接入后端私有网络的接口,则 eth1 上也配置有一个真实 IP 地址和一个浮动 IP 地址。在出现故障切换 Failover

的时候, 接入 Internet 的虚拟接口和接入私有网络的虚拟接口会同时切换到 Backup 的 LVS Router 上, 而为了不影响对 Internet 客户端的请求响应, 位于私有网络中的后端服务器均使用 NAT 路由的浮动 IP 作为与主 LVS Router 通信的默认路由。

在图 4-6 的架构中, 对外提供服务的公有 VIP (Public Virtual IP Address) 和私有 NAT VIP (NAT Virtual IP Address) 均被配置在物理网卡上, 而最佳的配置方式是将两个 VIP 各自配置到不同的物理网卡上, 即在这种配置下, 每个 LVS Router 节点最多只需两个物理网卡。在 NAT 路由转发中, 主 LVS Router 负责接收请求, 并将请求的目的地址替换成 LVS Router 的 NAT Virtual IP 地址, 再将其转发到选中的后端服务器上, 同时服务器处理后的应答数据也通过 LVS Router 将其地址替换成 LVS Router 的 Public Virtual IP 地址, 然后再转发给 Internet 客户端, 这个过程也称为 IP 伪装, 因为对客户端而言, 服务器的真实 IP 地址已被隐藏。

在 NAT 路由实现的负载均衡中, 后端服务器上可以运行各种操作系统, 即后端服务器上的操作系统类型并不影响 LVS Router 的 NAT 路由功能, 但是, 使用 NAT 路由方式存在的一个缺点是, LVS Router 在大规模集群部署中可能会是一个瓶颈, 因为 LVS Router 要同时负责进出双向数据流的 IP 地址替换。

2. DR 路由

相对于其他的负载均衡网络拓扑, DR (Direct Routing) 路由方式为基于 Keepalived 的负载均衡系统提供了更高的网络性能, DR 路由方式允许后端服务器直接将处理后的应答数据返回给客户端, 而无需经过 LVS Router 的处理操作, 如图 4-7 所示。通过仅允许访问请求进入 LVS Router 并进行路由转发到后端服务器, 而将服务器应答数据流避开 LVS Router 并直接发送给客户端的方式, DR 路由方案极大降低了 LVS Router 造成网络瓶颈的可能性。

在典型的 DR 路由负载均衡实现方案中, 主 LVS Router 通过 VIP 接受客户端访问请求并通过调度算法将请求转发到后端服务器上, 后端服务器处理请求并将处理结果直接发送给客户端, 即绕过 LVS Router。在 DR 路由负载均衡集群中, 可以在不过多影响整个集群网络性能的前提下, 不断增加后端服务器的数量以增加服务器的处理能力, 即在大规模集群中, DR 路由方式相对 NAT 路由, 对集群网络的性能影响更小, 而 LVS Router 成为集群网络瓶颈的可能性也更小。所以, 在基于 Keepalived 的负载均衡架构中, Keepalived 的最佳路由方式是 DR 路由, 即在配置 Keepalived 的路由方式时, 优先将其设置为 DR。

4.1.5 Keepalived 配置与使用

Keepalived 高可用负载均衡器的配置主要是编辑 Keepalived 的配置文件 `/etc/keepalived/keepalived.conf`。为了演示 Keepalived 负载均衡的配置使用, 本节采用两个独立服务器来作为前端 Keepalived 负载均衡器, 其中一台服务器作为主用负载均衡器 (LB1), 另一台服务器作为 Standby 负载均衡器一备 (LB2), 后端服务器池由四台运行 HTTP 服务的节点构成, 后端服务器位于同一个私有网络中, 其真实 IP 地址段为 192.168.1.20-192.168.1.23, 由

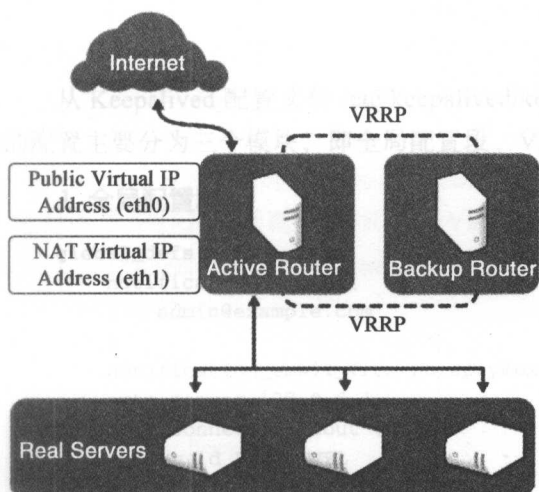


图 4-6 NAT 路由实现的 Keepalived 负载均衡

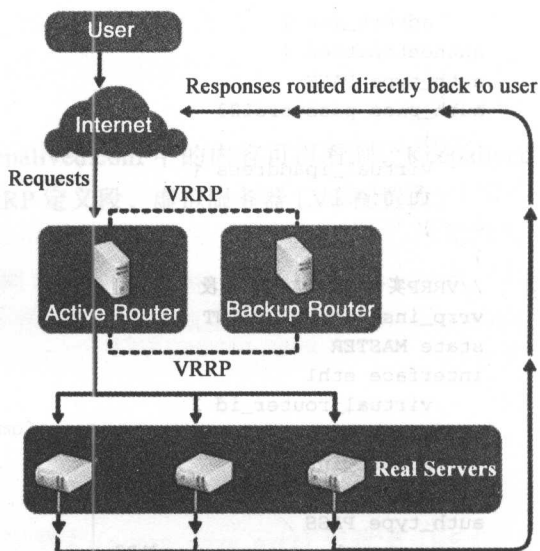


图 4-7 DR 路由实现的 Keepalived 负载均衡

Keepalived 控制的 VIP 为 10.0.0.1。此外，每个负载均衡器配有两张物理网卡 eth0 和 eth1，其中 eth0 接入 Internet，eth1 与后端服务器一起接入私有网络，此处的负载均衡器采用 Round-Robin 调度算法，由于后端节点数量很小，Keepalived 的路由方法可以设置为 NAT。从拓扑架构来看，这是与 4.1.2 节中的图 4-4 类似的典型二层负载均衡架构，要实现该二层负载均衡架构，LB1 中的 Keepalived 配置文件 /etc/keepalived/keepalived.conf 内容如下。

```
//全局配置段
global_defs {
    notification_email {
        admin@example.com
    }
    notification_email_from noreply@example.com
    smtp_server 127.0.0.1
    smtp_connect_timeout 60
    router_id LVS_DEVEL
}
//VRRP配置段
vrrp_sync_group VG1 {
    group {
        VRRP_EXT
        VRRP_INT
    }
}
//VRRP实例VRRP_EXT配置段
vrrp_instance VRRP_EXT {
    state MASTER
    interface eth0
        virtual_router_id 50
    priority 100
```

```

advert_int 1
authentication {
auth_type PASS
auth_pass password123
}
virtual_ipaddress {
10.0.0.1
}
}
//VRRP实例VRRP_INT配置段
vrrp_instance VRRP_INT {
state MASTER
interface eth1
virtual_router_id 2
priority 100
advert_int 1
authentication {
auth_type PASS
auth_pass password123
}
virtual_ipaddress {
192.168.1.1
}
}
//虚拟服务器LVS配置段
virtual_server 10.0.0.1 80 {
delay_loop 6
lb_algo rr
lb_kind NAT //NAT路由方式
protocol tcp
//后端服务器1
real_server 192.168.1.20 80 {
TCP_CHECK {
connect_timeout 10
}
}
//后端服务器2
real_server 192.168.1.21 80 {
TCP_CHECK {
connect_timeout 10
}
}
//后端服务器3
real_server 192.168.1.22 80 {
TCP_CHECK {
connect_timeout 10
}
}
//后端服务器4
real_server 192.168.1.23 80 {
TCP_CHECK {

```




```

        connect_timeout 10
    }
}

```

从 Keepalived 配置文件 `/etc/keepalived/keepalived.conf` 中的内容可以看到, Keepalived 的配置主要分为三个模块, 即全局配置段、VRRP 定义段、虚拟服务器 LVS 配置段。

1. 全局配置段

```

global_defs {
    notification_email {
        admin@example.com
    }
    notification_email_from noreply@example.com
    smtp_server 127.0.0.1
    smtp_connect_timeout 60
    router_id LVS_DEVEL
}

```

全局配置段 (`global_defs`) 的主要作用之一就是 Keepalived 出现故障时的邮件通知管理员, 让管理员以邮件形式知道 Keepalived 的运行情况。通常情况下, 邮件通知不是必须的, 用户可以选择其他监控方式来对 Keepalived 进行监控, 如 Nagios。需要说明的是, 全局配置段对 Keepalived 来说是可选的, 其内容并不是 Keepalived 配置所必须的。全局配置段的几个主要配置参数说明如下:

- ❑ `Notification_email`: 用于配置接收邮件的负载均衡器的管理员群组邮箱。
- ❑ `Notification_email_from`: 自定义发出邮件的邮箱地址, 即管理员邮件显示的发件人。
- ❑ `SMTP`: 指定简单邮件参数协议服务器地址, 一般为本机。
- ❑ `LVS_ID`: LVS 负载均衡器标志, 同一网络中其值唯一。

2. VRRP 配置段

```

vrrp_sync_group VG1 {
    group {
        VRRP_EXT
        VRRP_INT
    }
}

vrrp_instance VRRP_EXT {
    state MASTER
    interface eth0
        virtual_router_id 50
    priority 100
        advert_int 1
    authentication {
        auth_type PASS
        auth_pass password123
    }
    virtual_ipaddress {

```

```

10.0.0.1
}
}
vrrp_instance VRRP_INT {
state MASTER
interface eth1
    virtual_router_id 2
priority 100
    advert_int 1
authentication {
auth_type PASS
    auth_pass password123
}
    virtual_ipaddress {
        192.168.1.1
    }
}
}

```

VRRP 配置段 (vrrp_sync_group) 主要用于定义 VRRP 组, 在 Keepalived 发生任何状态变化时, 被定义在 VRRP 组中的 VRRP 实例作为逻辑整体一致行动, 如在发生 LVS Router 故障切换 Failover 的过程中, VRRP 组中的实例会作为一致整体同时切换。在本节的演示配置中, 同一个 VRRP 组内配置了两个 VRRP 实例, 分别是针对外部网络的 VRRP_EXT 实例和针对内部私有网络的 VRRP_INT 实例。VRRP 配置段中的关键参数说明如下。

- ❑ Vrrp_sync_group: VRRP 实例一致组, 用于定义 VRRP 一致组中的成员, 组内的 VRRP 实例行为是一致的, 如在 Failover 的时候, 一致组内的 VRRP 实例将同时迁移。在本机示例中, 当 LB1 出现故障时, VRRP_INT 和 VRRP_EXT 实例将同时切换到 LB2 上。
- ❑ Vrrp_instance: VRRP 实例, 用于配置一个 VRRP 服务进程实例, 其中的 State 设定了当前节点上 VRRP 实例的主备状态, 在主 LVS Router 中, 该值应该为 MASTER, 在备 LVS Router 中, 其值为 BACKUP。正常情况下, 只有 Master 的 LVS Router 在工作, Backup 的 LVS Router 处于 Standby 状态。
- ❑ Interface: 对外提供服务的网络接口, 如 eth0 和 eth1, 选择服务接口时, 一定要核实清楚, LVS Router 的 VIP 将会配置到这个物理接口上。
- ❑ Virtual_Router_id: 虚拟路由标志, 同一个 VRRP 实例使用唯一的标识。即同一个 VRRP 实例中, MASTER 和 BACKUP 状态的 VRRP 实例中, virtual_router_id 值是相同的, 同时在全部 VRRP 组内是唯一的。
- ❑ Priority: 此参数指明了该 VRRP 实例的优先级, 数字越大说明优先级越高, 取值范围为 0-255, 在同一个 VRRP 实例里, MASTER 的优先级高于 BACKUP。若 MASTER 的 Priority 值为 100, 那么 BACKUP 的 Priority 只能是 90 或更小的数值。
- ❑ Advert_int: Master 路由发送 VRRP 广播的时间间隔, 单位为秒。

❑ **Authentication**：包含验证类型和验证密码，类型主要有 PASS 和 AH 两种，通常使用的类型为 PASS。验证密码为明文，同一 VRRP 实例 MASTER 与 BACKUP 使用相同的密码才能正常通信。

❑ **Virtual_ipaddress**：虚拟 IP 地址，即 VIP，可以有多个虚拟 IP 地址，每个地址占一行，不需要指定子网掩码。

作为 Standby 的负载均衡器，LB2 的 `keepalived.conf` 配置文件与 LB1 类似，其不同之处在于 VRRP 实例配置段中的 VRRP 实例 State 和 Priority 参数的设置，如 LB1 中的 State 为 Master，LB2 中的 State 为 BACKUP，并且 LB2 中 VRRP 实例的 Priority 必须小于 LB1 中的优先级。

3. 虚拟服务器 LVS 配置段

```
virtual_server 10.0.0.1 80 {
    delay_loop 6
    lb_algo rr
    lb_kind NAT
    protocol tcp
    real_server 192.168.1.20 80 {
        TCP_CHECK {
            connect_timeout 10
        }
    }
    real_server 192.168.1.21 80 {
        TCP_CHECK {
            connect_timeout 10
        }
    }
    real_server 192.168.1.22 80 {
        TCP_CHECK {
            connect_timeout 10
        }
    }
    real_server 192.168.1.23 80 {
        TCP_CHECK {
            connect_timeout 10
        }
    }
}
```

虚拟服务器（Virtual Server）配置段主要定义 LVS 的监听虚拟 IP 地址和对应的后端服务器及其健康检测机制，虚拟服务器的定义段是 Keepalived 框架最重要的部分，也是其配置文件 `keepalived.conf` 中必不可少的部分。此部分的定义主要分为一个 Virtual Server 的定义和多个 Real Servers 的定义，Virtual Server 由 VRRP 中定义的 VIP 加上端口号构成，而 Real Server 由后端服务器节点 IP 和端口号构成，相关的配置参数说明如下。

❑ **Delay_Loop**：健康检查的时间间隔，单位为秒。

- ❑ LB_Algo: 选用的负载均衡算法, 示例中的 rr 表示 Round-Robin 算法。
 - ❑ LB_Kind: 采用的路由方法, 示例中采用的是 NAT 路由, 还可以采用 DR 和 TUN 路由。
 - ❑ Protocol: 转发协议, 一般有 TCP 和 UDP 两种。
 - ❑ TCP_CHECK: 表示采用 TCP 连接对 Real Servers 进行健康检查。
 - ❑ Connect_timeout: TCP 连接允许中断的时间, 单位为秒, 超过此值认为 TCP 连接 Timeout, 即后端服务器不可用。
- 上述示例中 Keepalived 的配置采用的是 NAT 路由方式, 而在大规模负载均衡集群中, NAT 路由通常造成网络性能瓶颈, 因此建议采用 DR 路由方式。DR 路由方式的配置与 NAT 方式类似, 如下 Keepalived 的配置将 Keepalived 配置成为提供后端服务器上 HTTP 服务 (80 端口) 的负载均衡器, 为了使用 DR 路由, 将 LB_Kind 参数配置为 DR, 其他的配置与 NAT 方式类似:

```
.....
virtual_server 172.31.0.1 80
    delay_loop 10
lb_algo rr
lb_kind DR                      //DR路由方式
persistence_timeout 9600
protocol TCP
.....
}
```

在 LB1 和 LB2 上配置完 keepalived.conf 后, 分别在两个节点上启动 Keepalived 服务, 即可正常使用 Keepalived 的负载均衡功能。

```
//启动Keepalived服务
systemctl start keepalived.service
//将Keepalived服务设置为开机启动
systemctl enable keepalived.service
```

4.2 HAProxy 概述与配置

4.2.1 HAProxy 概述

HAProxy 是由 Willy Tarreau 开发的一款具备高可用性、负载均衡、虚拟主机支持以及基于 TCP 和 HTTP 的应用代理开源软件, 基于 HAProxy 的负载均衡架构是最为常见的免费、快速且具备可靠性的集群负载均衡架构解决方案。此外, HAProxy 特别适合应用于需要会话保持或七层处理的高负载 Web 站点, 如淘宝、Twitter、Instagram、Github 和 Amazon 等大型网站都是 HAProxy 的使用者。就当前常见的硬件体系架构, 基于 HAProxy 的负载均衡系统完全可以支撑数以万计的并发连接, 同时, HAProxy 的运行模式使得将其整合到用户当前的基础架构中是个非常简单且安全的过程。通过 HAProxy 的代理, 还可以

避免用户的 Web 服务器直接暴露到外部网络中, 图 4-8 为典型的基于 HAProxy 的负载均衡集群拓扑架构, 图 4-9 为淘宝图片处理与存储内容分发网络 (Content Delivery Network, CDN) 系统所采用的 HAProxy 负载均衡架构。

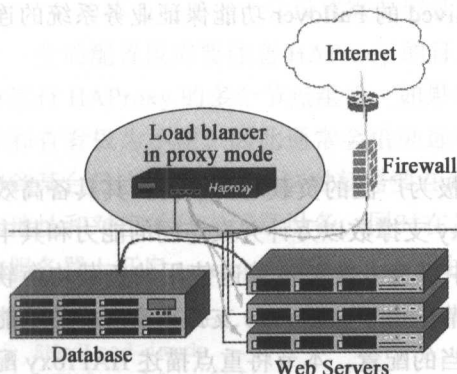


图 4-8 基于 HAProxy 的负载均衡拓扑架构

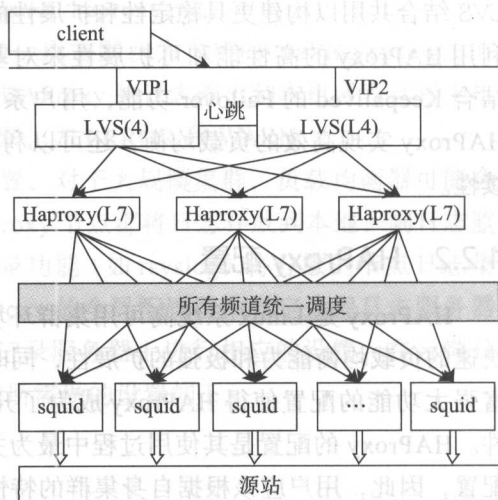


图 4-9 HAProxy 在淘宝 CDN 系统中的应用

HAProxy 实现的是一种事件驱动、单一进程的架构模型, 此类模型的优点在于能够支撑高并发大规模的连接。反之, 多进程或多线程模型受内存和系统调度器的限制以及无处不在的锁限制, 很难应对数以万计的高并发连接。HAProxy 支持连接拒绝, 通过拒绝连接, 可以限制某些非法或有意的攻击型连接, 从而降低其对网站带来的危害。HAProxy 的这一功能已成为目前应对小型 DDoS 攻击的主要方法之一, 并且其他负载均衡器很难做到这点。此外, HAProxy 还支持全透明代理, 即可以将客户端 IP 地址或者任何指定地址直接连接到后端服务器, 通过全透明代理, 可以不用修改某些特殊服务器地址而使其直接接收并处理部分特定流量。

HAProxy 主要为基于 HTTP 和 TCP 访问的应用服务提供负载均衡, 如基于 Internet 的连接服务和基于 Web 的应用服务。通过负载均衡算法, HAProxy 能够接受数以万计的访问请求并将其转发到后端服务器池中进行处理, 而后端服务器池就如一个强大的虚拟服务器接受 HAProxy 转发的请求并进行处理。HAProxy 的请求调度器 (Scheduler) 决定了后端服务器中每个服务器接受和处理的请求量, 在没有权重的调度算法下, 调度器为每台服务器分配相同数量的请求, 而在加权调度算法下, 调度器根据每台服务器的权重为每个后端服务器分配不同数量的请求。HAProxy 允许用户自定义多个代理, 并为每个代理提供负载均衡服务, 代理由一个前端和一个或多个后端构成, 前端定义了代理监听的 IP 地址 (Virtual IP) 和端口, 同时还需在前端定义中关联与其相关的后端, 而在 HAProxy 中, 后端主要用于定义服务器池和负载均衡算法。HAProxy 的负载均衡服务在 7 层, 即应用层, 在很多情况下, 由于商业应用连续性的要求, 管理员通常需要部署 HAProxy, 从而为基于 HTTP 的

应用提供负载均衡和高可用性。

4.1 节介绍的 Keepalived 也具有负载均衡的功能,但是相比而言,HAProxy 才是一个更为专业高效的负载均衡代理软件,在大多数情况下,用户通常将 HAProxy 和 Keepalived/LVS 结合共用以构建更具稳定性和扩展性的高可用环境,如淘宝的 CDN 架构便是如此。在利用 HAProxy 的高性能和可扩展性来对基于 HTTP 和 TCP 的服务进行负载均衡的同时,结合 Keepalived 的 Failover 功能,用户系统在应对大规模访问请求时候,不仅可以通过 HAProxy 实现高效的负载均衡,还可以利用 Keepalived 的 Failover 功能保证业务系统的连续性。

4.2.2 HAProxy 配置

HAProxy 是 Linux 系统高可用集群环境中使用极为广泛的负载均衡软件,其具备高效快速的负载均衡能力和极强的扩展性,同时 HAProxy 支撑数以万计并发访问的能力和其丰富强大功能的配置使得 HAProxy 成为了开源领域中极为专业并被普遍使用的负载均衡软件。HAProxy 的配置是其使用过程中最为关键的环节,由于 HAProxy 支持丰富多样的功能配置,因此,用户应该根据自身集群的特性进行适当的配置,本节将重点描述 HAProxy 配置中最为基础常见的配置。

HAProxy 的配置文件是 /etc/haproxy/haproxy.cfg,该文件是 HAProxy 功能配置的集中文件,其代理和负载均衡功能的配置均位于该配置文件中。HAProxy 的配置文件主要分为四个部分,即全局功能配置段、默认属性配置段、前端代理配置段、后端负载均衡配置段,各个配置段常见属性设置和功能描述如下。

1. 全局配置段

全局配置段的参数将被应用到全部运行 HAProxy 的节点中,全局配置段并不针对具体的代理和负载均衡进行设置,其需要配置的参数也相对简单,典型的 haproxy.cfg 全局配置段内容如下:

```
global
daemon
maxconn 4000
pidfile /var/run/Haproxy.pid
user Haproxy
group Haproxy
stats socket /var/lib/Haproxy/stats
log 127.0.0.1 local0
```

上述 HAProxy 的全局配置段中,用户为 HAProxy 常用的全局变量配置了参数,这些参数通常是进程级别并与操作系统相关的参数,并在全局上限定了 HAProxy 的工作特性,几个重要的参数解释如下。

❑ **Daemon:** 指定 HAProxy 以后台进程的形式运行。

- ❑ Group: 运行 HAProxy 的用户属组, 此处为 HAProxy。
- ❑ Maxconn: HAProxy 代理允许的最大并行连接数, 此处为 4000, 默认为 2000。
- ❑ User: 运行 HAProxy 的用户, 此处为 HAProxy。
- ❑ Pidfile: HAProxy 的进程文件, 此处为 /var/run/haproxy.pid。
- ❑ Log: 设置 HAProxy 运行日志的输出设备, 通常默认为本机的 /var/log/syslog, 并默认记录 INFO 级别的日志, 用户可以将 HAProxy 的日志输出到本机或者远程主机的日志设备上, 并设置需要记录的日志级别, 如 ERROR 和 WARN。

全局配置段需要注意 HAProxy 的日志设备配置, 对于大规模集群, 负载均衡器可能会由运行 HAProxy 的多个节点组成, 如果每个 HAProxy 节点都将日志存放到本地, 则日志监控和查看极为不便, 因此通常会借助远程日志记录功能 (如 rsyslog) 将分散的节点日志集中到某台日志服务器上, 这时就需要在每个 HAProxy 的全局配置段中指定远程日志服务器的地址和对应的日志记录设备, 同时在远程日志记录服务器上进行相应的设置。在云端日志服务器上开启 rsyslog 的 HAProxy 日志记录功能所需做的设置如下:

```
//将/etc/rsyslog.conf中的如下两行注释取消
$ModLoad imudp
$UDPServerRun 514
//为HAProxy设置日志设备和输出文件, 这里定义的日志设备要与haproxy.cfg全局配置段中log参数设置
  的匹配, 在/etc/rsyslog.conf中添加如下语句:
local0.*                /var/log/haproxy/haproxy.log
//在/etc/sysconfig/rsyslog中修改SYSLOGD_OPTIONS的值为如下:
SYSLOGD_OPTIONS="-r -m 0 -c 2"
//分别重启rsyslog和haproxy
systemctl restart rsyslog.service
systemctl restart haproxy.service
//跟踪haproxy的日志
tail -f /var/log/haproxy/haproxy.log
```

2. 默认配置段

默认 (default) 配置段设置的参数会被 haproxy.cfg 的其他配置段继承, 如 frontend、backend 和 listen 配置段都会继承 default 配置段参数, 同时这些配置段也可以写 default 配置段的参数值, 通常情况下, 用户可以将具有共性的参数放到 default 段进行统一配置, 然后再到各个配置段中进行个性修改, 常见的 default 配置段如下:

```
defaults
mode                http
log                 global
option              httplog
option              dontlognull
retries              3
timeout http-request 10s
timeout queue        1m
timeout connect      10s
timeout client        1m
```

```
timeout server 1m
```

默认配置段主要配置参数的解释如下：

- ❑ **Mode**：指定 HAProxy 实例使用的连接协议，即源请求到后端服务器之间的连接协议，可能值为 HTTP 和 TCP。对基于 HTTP 的 Web 应用服务，通常使用 HTTP 模式，对于其他应用服务，通常使用 TCP 模式。
- ❑ **Log**：指定日志地址和记录日志条目的 syslog/rsyslog 日志设备，此处的 global 表示使用 global 配置段中设定的 log 值。
- ❑ **Option**：日志记录选项，httplog 表示记录与 HTTP 会话相关的各种属性值，包括 HTTP 请求、会话状态、连接数、源地址以及连接时间等。dontlognull 表示不记录空会话连接日志，即 HAProxy 不会记录没有数据传输的会话连接日志，基于互联网的 Web 应用中不推荐使用 dontlognull，因为很多空会话连接可能包含有恶意行为，如恶意的端口漏洞扫描就是一种没有数据传输的空连接。
- ❑ **Retries**：在连接失败后重新尝试请求连接的次数，超过设置的尝试次数将认为连接失败。
- ❑ **Timeout**：设置各种请求、连接、响应的 Timeout 时间，单位为秒 (s) 或者毫秒 (m)。
- ❑ **HTTP-Request**：等待客户端完整 HTTP 请求的时间，此处为等待 10s。
- ❑ **Queue**：设置删除连接和客户端收到 503 或服务不可用等提示信息前的等待时间，此处等待时间为 10 毫秒。
- ❑ **Connect**：设置等待服务器连接成功的时间，此处为等待 10s。
- ❑ **Client**：设置允许客户端处于非活动状态，即既不发送数据也不接收数据的时间，此处为 1 毫秒。
- ❑ **Server**：设置服务器超时时间，即允许服务器处于既不接收也不发送数据的非活动时间，此处为 1 毫秒。

3. Frontend 配置

前端配置主要完成两个功能：一是配置监听客户端请求的 IP 地址和端口，在高可用环境下，此处的监听 IP 地址通常为虚拟 IP；二是将监听到的客户端请求转发到指定的后端配置中进行负载均衡。典型的 HAProxy 前端配置如下。

```
frontend WEB
bind 192.168.0.10:80
default_backend app
```

HAProxy 中允许配置多个前端，前端名称可以自定义，此处设置了名为 Web 的 HAProxy 前端，通过 bind 参数指定该前端监听的 IP 为 192.168.0.10，端口为 80，而该前端对应的后端名称是 app，一旦 Web 前端监听到连接，就会将该连接直接转给名为 app 的后端处理。前端与后端是 HAProxy 配置中的关键部分，通常前端负责监听请求连接，后端负责负载均衡。在 OpenStack 高可用集群配置中，由于要为每个 OpenStack 服务进行高可用配

置, 因此最佳做法就是将每个服务配置为一个前端和后端的组合, 并将前端监听的 VIP 通过 Pacemaker 或者 Keepalived 进行高可用设计。基于 HAProxy 的 OpenStack 高可用集群前端配置示例如下。

```
//数据库服务前端
frontend vip-db
bind 192.168.142.201:3306
timeout client 90m
default_backend db-vms-Galera
//qpid消息服务前端
frontend vip-qpid
bind 192.168.142.215:5672
timeout client 120s
default_backend qpid-vms
//dashboard 服务前端
frontend vip-horizon
bind 192.168.142.211:80
timeout client 180s
cookie SERVERID insert indirect nocache
default_backend horizon-vms
```

4. Backend 配置

后端配置主要实现两个主要功能: 一是负载均衡调度算法的设置; 二是设置最终响应请求的服务器池各个节点的 IP 地址和端口, 并设置每个节点的健康检查方式。一个典型的后端配置如下。

```
backend app
balance roundrobin
server app1 192.168.1.1:80 check
server app2 192.168.1.2:80 check
server app3 192.168.1.3:80 check inter 2s rise 4 fall 3
server app4 192.168.1.4:80 backup
```

HAProxy 允许配置多个后端, 并且每个后端都有一个前端与其对应, 后端实例的名称可以由用户自定义, 但是一定要与对应前端中设置的后端名称一致。上述后端配置中, 后端实例的名称是 app, 采用 Round-Robin 负载均衡算法, sever 行定义后端的真实服务器, 服务器的名称为 app1、app2、app3 和 app4, 这里的服务器名称并非真实的后端服务器主机名, 而只是便于识别的自定义服务器名称, 服务器的具体地址通过紧随其后的 IP 地址和端口号来确定。此外, 在定义后端服务器的同时, 通过 check 参数还可指定 HAProxy 对服务器的健康检查方式, 上述配置中, 后端服务器 app3 中的 inter 2s 指定了对 app3 进行健康检查的时间间隔是 2s, rise 4 表示 HAProxy 对 app3 发起 4 次健康检查均正常则认为 app3 正常, fall 3 表示连续 3 次健康检查失败则认为 app3 已经故障。

HAProxy 后端配置中指定了负载均衡所采用的算法, HAProxy 支持多种负载均衡算法, 用户可以根据后端服务器池中各个节点的实际资源配置进行不同的算法选取, HAProxy 支持的负载均衡算法有以下几种。

- ❑ Round-Robin (round robin): 与 Keepalived 的 Round-Robin 类似, 使用这种算法, 服务请求会被轮询转发到服务器池中的每一个服务器上, 而不去评估服务器的当前负载和处理能力, 服务器池中的每个节点都被轮询转发请求。
- ❑ Static Round-Robin (static-rr): 与 Round-Robin 一样轮询转发请求到每一个后端服务器, 但是不允许对后端服务器进行动态加权设置, 即服务器的权重是静态固定的, 而由于权重静态固定, 后端服务器池中的节点数目不受限。
- ❑ Least-Connection (leastconn): 即最少连接数算法, 与 Keepalived 的最少连接数算法类似, 后端服务器活动连接数越多, 则接收到的服务请求就越少, 反之, 则接收到的服务请求越多。
- ❑ Source (source): 该算法将请求中的源 IP 地址进行 HASH 后除以全部正常运行的后端服务器权重来决定接收服务请求的服务器, 在这种算法中, 同一个客户端 (相同的源 IP 地址) 发出的请求会被固定转发给某一个固定的后端服务器。但是, 如果服务器权重大小发生改变或者服务器数目出现变动, 则响应该客户端请求的后端服务器会改变, 因为这时的 HASH/Wight 值已经改变。
- ❑ URL (url): 该算法将请求 URL 字符串进行 HASH 并除以全部正常运行的后端服务器权重来决定接收服务请求的服务器, 在这种算法中, 指向同一目标站点的服务请求会被固定转发到相同的后端服务器上。URL 也称为基于目标地址的 HASH 负载均衡算法, 主要用于 Web Cache 集群中, 通过 URL 负载均衡算法, 可以避免请求因为指向不同的 Cache 服务器而导致缺页, 而缺页会导致刷新 Cache 最终降低系统响应速率。
- ❑ URL Parameter (url_param): 该算法通过查询源 HTTP 请求报文中的某一字符串参数并将其进行 HASH 后除以全部服务器权重来决定接收服务请求的服务器。如果 HTTP 报文中没有需要的参数, 则默认使用 Round-Robin 算法。
- ❑ Header Name (hdr): 该算法通过查询 HTTP 请求报文中的 HEAD 字段并将其进行 HASH 后除以全部服务器权重来决定接收服务请求的服务器。如果报文中没有 HEAD 参数, 则默认使用 Round-Robin 算法。

在高可用集群配置中, 为了实现服务的高可用, 通常每个后端配置中都需要提供两个以上的后端服务器进行负载均衡。在 OpenStack 高可用集群配置中, 为了实现每个 OpenStack 服务的高可用性, 通常为每个服务配置三个控制节点, 而这三个控制节点即 HAProxy 后端配置中的后端服务器, OpenStack 高可用集群配置的 HAProxy 后端配置示例如下。

```
//heat服务后端
backend heat-srv-vms
balance roundrobin
server controller1-vm 192.168.142.110:8004 check inter 1s
server controller2-vm 192.168.142.111:8004 check inter 1s
server controller3-vm 192.168.142.112:8004 check inter 1s
```



```
//ceilometer服务后端
backend ceilometer-vms
balance roundrobin
server controller1-vm 192.168.142.110:8777 check inter 1s
server controller2-vm 192.168.142.111:8777 check inter 1s
server controller3-vm 192.168.142.112:8777 check inter 1s
//MySQL数据库服务后端
backend db-vms-Galera
option httpchk
option tcpka
stick-table type ip size 1000
stick on dst
timeout server 90m
server controller1-vm 192.168.142.110:3306 check inter 1s port 9200 backup on-
marked-down shutdown-sessions
server controller2-vm 192.168.142.111:3306 check inter 1s port 9200 backup on-
marked-down shutdown-sessions
server controller3-vm 192.168.142.112:3306 check inter 1s port 9200 backup on-
marked-down shutdown-sessions
.....
```

HAProxy 配置完成之后, 在各个节点启动 HAProxy 服务即可正常使用 HAProxy 强大的负载均衡功能。

```
//启动haproxy服务
systemctl start haproxy.service
//设置开机自启动haproxy服务
systemctl enable haproxy.service
```

4.2.3 HAProxy 监控页面

HAProxy 为每个监听代理提供了实时监控, 并可以将监控参数以 GUI 页面的形式呈现给用户。要使用 HAProxy 的 GUI 页面, 需要在 `/etc/haproxy/haproxy.cfg` 配置文件中配置相应的监听参数, 通常需要配置一个 Listen 配置段 (也可以是 Frontend 或 Backend 配置段), 即可通过 HTTP 协议访问 HAProxy 的监控页面, 最为常用的 HAProxy 监控页面配置如下。

```
//定义一个listen, 也可以放在frontend或backend段中
listen stats
//使用协议
mode http
//监听地址和端口
bind 192.168.142.110:8080
//启用信息统计功能
stats enable
//隐藏版本号
stats hide-version
//访问URL
stats uri /HAproxy?openstack
```

```
//登录提示信息
stats realm      HAproxy\Statistics
//admin界面，验证成功后允许管理节点
stats admin if    TRUE
//登录用户名和密码
stats auth        admin:admin
//页面刷新时间
stats refresh     10s
```

通过上述 Listen 段的配置，在重启 HAProxy 之后即可在浏览器上登录访问 HAProxy 的监控页面。根据上述配置，访问地址为 `http://192.168.142.110:8080/HAproxy?openstack`，访问的用户密码均为 `admin`，监控页面每隔 10s 刷新一次，为了安全起见，上述配置还隐藏了 HAProxy 的版本号，登录成功后的 HAProxy 监控页面如图 4-10 所示，HAProxy 默认通过不同的颜色来区别各个前端 IP 和后端服务器的运行状态，HAProxy 配置文件中对应的每个前端和后端配置段都会有一个独立的统计表格，通过该统计表可以很清楚地了解和

分析各个服务的运行状况。图 4-10 中，并未出现 HAProxy 的版本信息，这是因为配置了 HAProxy 的版本隐藏选项（`stats hide-version`），由于版本信息通常会暴露特定开源软件版本的漏洞，因此从安全角度考虑不建议将开源软件版本暴露给别人。

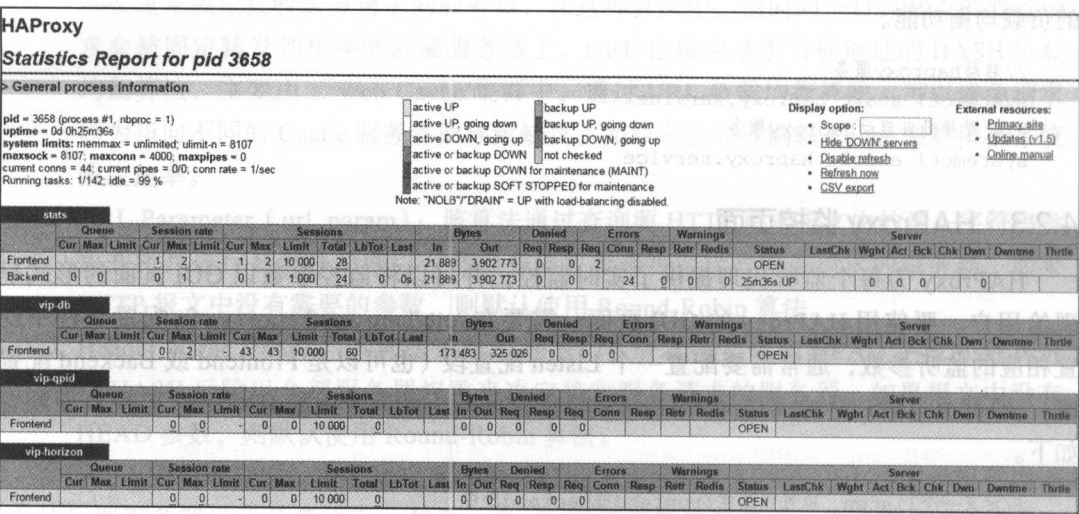


图 4-10 HAProxy 监控页面

HAProxy 的监控页面将每项资源的监控参数以表格形式呈现给用户，并将监控参数划分为七个类别，即 Queue、Session rate、Sessions、Bytes、Denied、Errors、Warning、Server，每组参数类别下又有多个详细参数，其中各个参数的解释如下。

(1) Queue

- ❑ Cur: 表示当前队列的请求数量。
- ❑ Max: 表是当前队列最大的请求数量。

□ Limit: 表示队列的限制数量。

(2) Session rate

□ Cur: 每秒会话连接数量。

□ Max: 每秒会话数量最大值。

□ Limit: 每秒会话数量的限制值。

(3) Sessions

□ Total: 总共会话数量。

□ Cur: 当前的会话数量。

□ Max: 最大会话数量。

□ Limit: 会话连接限制。

□ Lbtot: 选中一台服务器所用的总时间。

□ Last: 最后一次会话时间。

(4) Bytes

□ In: 网络会话输入字节数总量。

□ Out: 网络会话输出字节数总量。

(5) Denied

□ Req: 被拒绝的会话请求数量。

□ Resp: 拒绝回应的请求数量。

(6) Errors

□ Req: 错误的请求数量。

□ Conn: 错误连接数量。

□ Resp: 错误响应数量。

(7) Warnings

□ Retr: 重新尝试连接的请求数量。

□ Redis: 重新发送的请求数量。

(8) Server

□ Status: 后端服务器状态, 可以有 UP 和 DOWN 两种状态。

□ LastChk: 持续检查后端服务器的时间。

□ Wght: 服务器权重。

□ Act: 活动后端服务器数量。

□ Bck: 后端备份服务器的数量。

□ Down: 状态为 Down 的后端服务器数量。

□ Downtime: 服务器总的 Downtime 时间。

□ Throttle: 状态 Backup 变为 Active 的服务器数量。

4.2.4 HAProxy 配置参考

作为一个专业的负载均衡代理软件，HAProxy 具有非常丰富的配置选项，4.2.2 节介绍的 HAProxy 配置属于较为常见和通用的配置，除此之外，HAProxy 还有很多详细功能配置项^①，为了满足不同应用场景下用户的 HAProxy 配置参考，本节将会对 HAProxy 的配置文件 haproxy.cfg 中各个配置段的配置参数，以及这些参数的功能作用进行解释和总结，用户可以根据自己的需求在相应配置段中进行参数取舍，HAProxy 各个配置段示例如下。

1. 全局配置段参考

```
global
//默认最大连接数
maxconn 20000
//level为err/warning/info/debug日志记录设备
log $host_ip local0
//chroot运行的路径
chroot /var/HAproxy
//HAproxy进程属组UID
uid 200
//HAproxy进程属组GID
gid 200
//以后台形式运行HAproxy
daemon
//进程数量，可以设置多个进程提高性能
nbproc 2
//HAproxy的pid存放路径
pidfile /var/run/HAproxy.pid
//ulimit的数量限制
ulimit-n 65535
```

2. 默认配置段参考

```
defaults
//使用global定义的日志记录设备
log global
//设置实例的运行模式或协议，当实现内容交换时，前端和后端，必须工作在同一种模式
mode {tcp|http|health}
//最大连接数
maxconn 20000
//日志类别为http日志格式
option httplog
//每次请求完毕后主动关闭http通道
option httpclose
//不记录健康检查的日志信息
option dontlognull
//允许在发往服务器的请求头部中插入 “X-Forwarded-For” 头部
```

① <http://cbonte.github.io/HAproxy-dconv/configuration-1.4.html#5>

```

option forwardfor
//serverId对应的服务器挂掉后, 强制定向到其他健康的服务器
option Redispatch
//允许结束掉当前队列中一直pending的连接
option abortonclose
    //统计页面刷新间隔
    stats refresh 30
//认为服务不可用的尝试连接次数
retries 3
//默认的负载均衡的方式, 这里为轮询方式, 也可以是balance source, 即IP hash方式; 或者balance
    leastconn, 即最小连接方式
balance roundrobin
//连接超时
timeout 5000
//客户端超时
clitimeout 50000
//服务器超时
srvtimeout 50000
//心跳检测超时
timeout check 2000

```

3. 监控页面配置段参考

```

//对Frontend和Backend进行监控统计, 监控组名称可以自定义
listen admin_status
//监听IP地址及端口
bind 10.0.0.1:65532
    //http的7层模式
mode http
//仅记录错误到本机的local0日志设备上
log 127.0.0.1 local0 err
//每隔10秒自动刷新一次监控页面
stats refresh 10s
//登录监控页面的url
stats uri /admin?stats
//监控页面的提示信息
stats realm itnihao\ itnihao
//监控页面的用户和密码admin, 可以设置多个用户名
stats auth admin:password
//隐藏统计页面上的HAproxy版本信息, 防止黑客针对版本进行恶意攻击
stats hide-version
//下面语句用于自定义访问页面出错时候, 呈现给用户对应错误代码的页面内容及该页面内容的位置, 配置格式为: errorfile <code> <file>, 支持代码包括200、400、403、408、500、502、503和504
errorfile 200 /etc/HAproxy/errorfiles/200.http
errorfile 400 /etc/HAproxy/errorfiles/400.http
errorfile 502 /etc/HAproxy/errorfiles/502.http
errorfile 503 /etc/HAproxy/errorfiles/503.http
errorfile 504 /etc/HAproxy/errorfiles/504.http
//下面语句用于捕获并将指定的请求/响应首部记录到HAproxy的日志中, 日志中记录的是指定首部的值
capture request header Host len 40
capture request header Content-Length len 10

```



```
capture request header Referer len 200
capture response header Server len 40
capture response header Content-Length len 10
capture response header Cache-Control len 8
```

4. FrontEnd 配置段参考

Frontend 配置段主要通过 bind 配置监听的虚拟 IP 地址和端口，同时 Frontend 配置里面可以定义多个 acl 以进行请求精确匹配，Frontend 配置段中还可以定义与全局默认配置段重名的参数以覆盖全局配置段的参数。

```
//定义前端名称，名称可以由用户自定义
frontend web_server
//监听的IP地址和端口，通常监听的IP为虚拟IP
bind 0.0.0.0:80
//使用http协议
mode http
//应用全局的日志配置
log global
//启用http的log
option httplog
//每次请求完毕后主动关闭http通道，HA-Proxy不支持keep-alive模式
option httpclose
//允许在发往服务器的请求头部中插入“X-Forwarded-For”头部
option forwardfor
//下面语句配置了acl策略来请求匹配，如果请求的域名满足www.warrior.cn，则返回true，参数-i表示
//匹配忽略大小写：
acl warrior_blog hdr_dom(host) -i blog.warrior.cn
//下面的语句表示对ACL匹配结果的不同响应处理方式，如果warrior_blog的acl匹配返回值为true，则阻
//止请求，返回错误代码403
block if warrior_blog
//如果warrior_blog的acl匹配返回值为true，则使用名为warrior_web的后端
use_backend warrior_web if warrior_blog
```

5. BackEnd 配置段参考

Backend 配置段主要配置负载均衡算法，定义后端服务器以及相应的健康检查方式等参数，同时 Backend 配置段也可以定义与默认全局配置段重名的参数，从而覆盖全局参数值以进行局部后端自定义。

```
backend server_web
//http的7层模式
mode http
//负载均衡的方式，roundrobin平均方式
balance roundrobin
//在某条件下拒绝持续连接，适用于对静态文件的负载均衡。
option ignore-persist { if | unless } <condition>
//启用双向超时处理，如socket的read和write
option independant-streams
//记录健康检查日志
option log-health-checks
```

```

//对非完全成功的连接改变日志记录等级
option log-separate-errors
//传输大文件时可以提前记录日志
option logasap
//MySQL健康检查
option mysql-check
//强制将http请求发往已经down掉的server
option persist
//是否允许重新分配在session失败后
option redispatch
//smtp检查
option smtpchk
//通过http协议进行健康检查
option httpchk
//允许对单个socket进行统计
option socket-stats
//是否允许向server 发送keepalive
option srvtcpka
//是否允许向server和client发送keepalive
option tcpka
//允许记录tcp连接的状态和时间
option tcplog
//允许客户端透明代理
option transparent
//心跳检测的文件
option httpchk GET /lb.html HTTP/1.0
//为当前后端配置粘性表;表存储条目类型为IP地址,允许存储1k大小的IP地址。
tick-table type ip size 1024
//定义一个请求模式dst,以将一个客户端同某个后端服务器关联起来。
stick on dst
//后端服务器最大等待时间,超过此时间则认为服务器不可用。
timeout server 90m

```

如下进行多后端服务器定义,check inter 1500 是检测心跳频率, rise 3 表示 3 次检查结果正确则认为服务器可用, fall 3 表示检测结果失败 3 次则认为服务器不可用, weight 代表服务器权重。port 9200 表示通过端口 9200 来进行基于 http 的健康检查, backup 表示该服务器是备份服务器,只有在其他非 backup 服务器均不可用的情况下负载均衡器才会使用该后端服务器,默认情况下使用第一个标记为 backup 的后端服务器, upon-marked-down shutdown-sessions 表示当该服务器被认为是 shutdown 的时候,关闭全部与该服务器的请求连接。

```

server 192.168.51.78 192.168.151.78:80 check inter 1500 rise 3 fall 3 weight 1
    port 9200 backup on-marked-down shutdown-sessions
server 192.168.151.79 192.168.151.79:80 check inter 1500 rise 3 fall 3 weight 1
    port 9200 backup on-marked-down shutdown-sessions
server 192.168.151.80 192.168.151.79:80 check inter 1500 rise 3 fall 3 weight 1
    port 9200 backup on-marked-down shutdown-sessions

```

4.3 本章小结

本章主要介绍了目前最为流行的开源负载均衡软件 Keepalived 和 HAProxy，在 Keepalived 的介绍中，还对使用极为广泛的 LVS 和 VRRP 进行了介绍。在目前主流的 OpenStack 高可用集群部署方案中，OpenStack 服务高可用最为常见的实现方式便是 Keepalived 或 Pacemaker 与 HAProxy 的组合方案，因此本章介绍的负载均衡软件均是后续进行 OpenStack 高可用集群部署必须掌握的基础软件。通过本章的学习，读者应该对 Keepalived 和 HAProxy 软件的历史发展、工作原理、架构拓扑和基本配置都有了充分的了解。在实际配置过程中，本章还分别总结了常见的 Keepalived 和 HAProxy 配置以供用户参考。

集群消息队列系统

OpenStack 是由 Nova、Neutron、Cinder 等多个组件构成的开源云计算项目，各组件之间通过 REST 接口进行相互通信和彼此调用，而组件之间的 REST 接口调用，是建立在基于高级消息队列协议（AdvancedMessageQueueProtocol，AMQP）上的 RPC 通信。在 OpenStack 中，AMQP Broker（可以是 Qpid Broker，也可以是 RabbitMQ Broker）位于 OpenStack 的任意两个内部组件之间，并使得 OpenStack 内部组件以松耦合的方式进行通信。纵观 OpenStack 的组件架构设计，其中的消息队列在 OpenStack 全局架构中扮演着至关重要的组件通信作用，也正是因为基于消息队列的分布式通信技术，才使得 OpenStack 的部署具有灵活、模块松耦合、架构扁平化和功能节点弹性扩展等特性，所以消息队列在 OpenStack 的架构设计和实现中扮演着极为核心的消息传递作用。同时，消息队列系统的消息收发性能和消息队列的高可用性也将直接影响 OpenStack 的集群性能。可以说，没有消息队列系统或者消息队列故障后的 OpenStack 集群将失去统一协调能力，整个集群将会支离破碎，各个组件“各自为政”，使得集群不具备任何云计算的能力。

对任何分布式集群系统而言，组件之间的耦合与彼此之间的交互都需要依赖消息通信系统，OpenStack 作为由诸多项目组件构成的分布式 IaaS 架构，其内部组件之间的彼此通信是整个 OpenStack 集群能够协调运行的根本。在 OpenStack 的部署过程中，用户可以选择不同的消息队列系统来为 OpenStack 提供消息服务，但是在众多的消息队列系统中，RabbitMQ 是使用最多也是综合性能最接近生产系统要求的消息队列系统。因此，本章将主要以 RabbitMQ 消息队列系统为主，讲解 RabbitMQ 的安装、部署与配置。同时，在 OpenStack 高可用集群部署中，作为最基础与核心的消息队列系统，RabbitMQ 提供的消息服务也必须具备高可用性。因此，本章还将从高可用性的角度，对 RabbitMQ 的高可用配置进行讲解。在介绍 RabbitMQ 之前，让我们先来了解下高级消息队列协议（AMQP）。

5.1 AMQP 概述

随着分布式集群 IT 系统架构的兴起, 异步消息模型被普遍用于不同组件之间的通信, 与此同时, 各种消息中间件产品及其相应的协议也随之出现并不断被完善。在这个过程中, 尽管同步消息处理机制领域已有众多的实现标准, 然而在异步消息实现上却没有统一标准可用, 而仅有一些大型商业公司具备自己的异步消息中间件产品, 如 Microsoft 的 MSMQ, IBM 的 WebsphereMQ 等。开源异步消息实现标准的缺失和不一致, 使得很多应用产品的选择在很大程度上受限于应用与不同中间件之间的耦合, 而且增加维护人员的工作和运维成本开销。为了解决商业异步消息队列产品的垄断封闭和成本问题, Cisco、Redhat 和 iMatix 等公司于 2006 年 6 月联合制定了异步消息队列标准, 即高级消息队列协议 AMQP, AMQP 是应用层协议的一个开放标准, 专为面向消息的中间件而设计, 同时, AMQP 也是面向消息、队列、路由、可靠性和安全性而设计的高级消息队列系统。

AMQP 提供统一消息服务的应用层标准协议, 客户端与消息中间件基于此协议传递消息, 并且消息传递不受不同客户端 / 中间件产品和不同开发语言等条件的限制。在消息的传递过程中, 消息中间件主要用于组件之间的解耦。基于消息中间件的解耦作用, 消息的发送者无须知道消息使用者的存在, 反之, 使用者也不关心消息的发送者是谁, 并且解耦过程所采用的协议与上层产品和程序语言无关。此外, AMQP 协议是一种二进制协议, 它让客户端应用与消息中间件之间异步、安全、高效地交互。从全局协议栈来看, AMQP 协议可划分为三层, 这种分层架构类似于 OSI 网络参考模型, 各层可独立替换实现而不影响与其他层的交互。在实现过程中, AMQP 通常会定义服务器端域模型, 以用于规范服务器的行为, AMQP 服务器端通常也称为 AMQP Broker。AMQP 的三层协议分别是 Model 层、Session 层和 Transport 层。

- ❑ Model 层: 模型层决定了基本域模型所产生的行为, 这种行为在 AMQP 中用“Command”表示。
- ❑ Session 层: 会话层定义客户端与 Broker 之间的通信, 通信双方都是一个 Peer, 可互称作 Partner, 会话层为模型层的 Command 提供可靠的传输保障。
- ❑ Transport 层: 传输层专注于数据传送, 并与 Session 保持交互, 接受上层的数据并组装成二进制流, 数据传送到 Receiver 后再进行解析并交付给 Session 层。Session 层需要 Transport 层完成网络异常情况的汇报, 顺序传送 Command 等工作。

另外, 在 AMQP 的定义及实现中, 有四个非常重要的概念, 分别为: 虚拟主机 (Virtual-Host), 交换机 (Exchange), 队列 (Queue) 和绑定 (Binding), 这几个概念的解释如下。

- ❑ 虚拟主机: 一个虚拟主机通常由一组交换机、队列和绑定构成。虚拟主机的作用主要是让用户细化权限控制, 如在 RabbitMQ 中, 用户只能以虚拟主机的粒度进行权限控制。因此, 如果需要禁止 A 组用户访问 B 组的交换机、队列和绑定, 则必须为 A 和 B 分别创建一个虚拟主机。在 RabbitMQ 中, 每个 Broker 都有一个默认的

根目录“/”虚拟主机。

- ❑ 队列：队列是 Messages 的终点，并由消费者创建。可以将队列理解成装消息的容器，正常情况下，消息应该一直保存在队列里，直到有客户端或消费者连接到这个队列并将 Message 取走为止。
- ❑ 交换机：交换机可以看成是具有路由功能的程序，每个消息都有一个路由键 (RoutingKey)，键值就是一个简单的字符串。交换机中有一系列的绑定，Binding 在 Queue 绑定到 Exchange 的时候设置，Binding 也称为 Exchange 的路由规则 (Route-Rule)。一个虚拟主机中可以有多个交换机，一个交换机可以与多个队列绑定，一个队列也可以绑定到多个交换机，即队列与交换机之间是多对多的关系。

除了上述三个主要的概念，在实现 AMQP 的过程中还需掌握很多相关术语并对其进行实现，以下是对 AMQP 中常见术语的简要描述。

- ❑ Broker：消息队列服务器实体，如 RabbitMQ-server 进程就是一个 Broker。
- ❑ Exchange：消息交换机，它指定消息按什么规则，路由到哪个队列。
- ❑ Queue：消息队列载体，用于存储消息的容器，每个消息都会被投入到一个或多个队列，Queue 需要绑定到 Exchange 上才能接受消息。
- ❑ Binding：绑定的作用就是把 Exchange 和 Queue 按照路由规则绑定起来。
- ❑ RoutingKey：路由关键字，Exchange 根据这个关键字与 Binding 的匹配关系来决定消息投递到哪个队列。
- ❑ VirtualHost：虚拟主机，一个 Broker 里可以创建多个虚拟主机，用于隔离不同用户的权限。
- ❑ Producer：消息的生产者，是投递消息的应用程序。
- ❑ Consumer：消息消费者，是接受消息的应用程序。
- ❑ Channel：消息通道，在客户端的每个连接里，可建立多个 Channel，每个 Channel 代表一个会话任务。

5.2 RabbitMQ 概述

RabbitMQ 用 Erlang 语言开发，是 AMQP 的开源实现。RabbitMQ 服务器端用 Erlang 语言编写，并支持多种客户端，如 Python、Ruby、.NET、Java、JMS、C、PHP、Action-Script、XMPP、STOMP 等。RabbitMQ 主要用于在分布式系统中存储和转发消息，在易用性、扩展性、高可用性等方面得到了普遍认同和使用。在 RabbitMQ 的部署使用和故障排除过程中，将会接触到很多 RabbitMQ 相关的术语，本节将重点介绍 RabbitMQ 的基础概念，了解和掌握这些概念，是使用 RabbitMQ 为集群系统提供消息服务的基础。

1. Connection 和 Channel

ConnectionFactory、Connection 和 Channel 都是 RabbitMQ 对外提供的 API 中最基本

的对象。Connection 是 RabbitMQ 的 Socket 连接，它封装了相关 Socket 协议的逻辑，简单地说，Connection 就是一个 TCP 的连接，Producer 和 Consumer 都通过 TCP 协议连接到 RabbitMQ Server，RabbitMQ 的初始化过程就是建立一个 TCP 连接的过程。ConnectionFactory 为 Connection 的制造工厂，即初始化 Connection 对象。Channel 是客户端与 RabbitMQ 交互消息最重要的一个接口，客户端大部分的业务操作是在 Channel 这个 API 接口中完成的，包括定义 Queue 和 Exchange、绑定 Queue 与 Exchange、发布消息等操作。Channel 也称为虚拟连接，它建立在 TCP 连接中，数据流动都是在 Channel 中进行，通常，程序启动后首先建立 TCP 连接，接着就是建立 Channel 对象。RabbitMQ 中使用 Channel 而不是直接使用 TCP 连接，根本的原因是创建和关闭 TCP 连接对系统来说开销很大，致使系统不能处理高并发的消息队列，然而使用基于 TCP 连接的虚拟 Channel 却可以解决这种因建立连接所需的额外系统资源开销。

2. Queue

Queue 是 RabbitMQ 的内部对象，用于存储消息，Queue 可以看成是存储消息的容器。RabbitMQ 中的消息只能存储在 Queue 中，生产者（图 5-1 中的 P）生产消息并最终投递到 Queue 中，消费者（图 5-1 中的 C）可以从 Queue 中获取消息并消费。

消息的生产者与消费者之间可以是一对多或者多对多的关系，即多个消费者可以订阅同一个 Queue（如图 5-2 所示），这时，Queue 中的消息会被平均分摊给多个消费者进行处理，而不是每个消费者都收到所有的消息并处理。



图 5-1 生产者与消费者一对一的 Queue

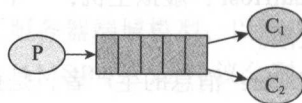


图 5-2 生产者与消费者一对多的 Queue

3. Message Acknowledgment

在实际应用场景中，可能会出现消费者收到 Queue 中的消息，但没有处理完就宕机或出现其他意外情况而导致消费者不能继续处理消息的情况。在这种情况下，未处理完成的消息可能会丢失。为了避免这种情况下的消息丢失，RabbitMQ 会要求消费者在处理完消息后发送一个回执应答给 RabbitMQ Broker，RabbitMQ 收到消息回执（Message-Acknowledgment）后才将该消息从 Queue 中移除。如果 RabbitMQ 没有收到回执并检测到消费者与 RabbitMQ Broker 的连接断开，则 RabbitMQ 会将该消息发送给其他消费者（如存在多个消费者）继续处理。RabbitMQ 的消息处理过程中并不存在 Timeout 的概念，即一个消费者处理消息所花的时间再长，只要其连接还存在，RabbitMQ 就不会将该消息发送给其他消费者。这种情况会产生另外一个问题，如果开发人员在处理完业务逻辑后，忘记发送回执给 RabbitMQ，这将会导致 Queue 中堆积的消息越来越多，而消费者重新连接到 RabbitMQ 后会重复消费这些消息并重复执行业务逻辑。因此，在处理完业务逻辑后一定要向 RabbitMQ 发送应答回执，否则会造成应用系统存在重大 BUG。

4. Message Durability

可以想象这样一种场景，在 Queue 中还有大量消息没有被消费者提取，此时 RabbitMQ 服务器出现故障或者其他意外情况导致 RabbitMQ 重启，而重启后 Queue 中的消息可能已经丢失。如果希望即使在 RabbitMQ 服务重启的情况下，消息也不会丢失，则可以将 Queue 与 Message 均做持久化设置，即 MessageDurability，这样便可保证绝大部分情况下 RabbitMQ 的消息不会丢失。但是，消息持久化方法依然解决不了小概率丢失消息事件的发生，如 RabbitMQ 服务器已经接收到生产者的消息，但还没来得及持久化该消息时 RabbitMQ 服务器就断电了，如果需要解决这种小概率事件下的消息丢失情况，那么就要用到事务 RabbitMQ 的高级功能或者将 RabbitMQ 部署为高可用集群。

5. Prefetch Count

如果有多个消费者同时订阅同一个 Queue 中的消息，Queue 中的消息会被平分给多个消费者。这时如果每个消费者处理消息的时间不同，就有可能导致某些消费者一直处于繁忙状态，而另外一些消费者因为处理能力较强而很快就处理完手头工作并一直处于空闲状态。要解决这一情况以提高整个消息系统的消息处理效率，可以设置预获取数目（PrefetchCount）来限制 Queue 每次发送给某个消费者的消息数，比如设置 PrefetchCount=1，则 Queue 每次给每个消费者发送一条消息，消费者处理完这条消息后 Queue 会再给该消费者发送下一条消息（如图 5-3 所示）。

6. Exchange

前文中提到的生产者将消息投递到 Queue 中，然而实际上 RabbitMQ 不会直接将消息投递到 Queue 中。消息投递过程实际上是生产者将消息发送到 Exchange（见图 5-4），由 Exchange 将符合转发规则的消息路由到一个或多个 Queue 中，而将不符合规则的消息直接丢弃。从功能实现上来看，Exchange 的功能就像一个路由器，符合路由规则的数据则转发到对应的目标地址，其他数据则被拒绝或者丢弃。

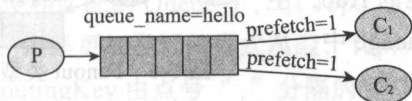


图 5-3 消费者预取消息队列

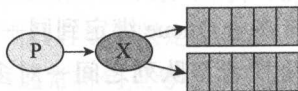


图 5-4 Exchange 交换队列

7. RoutingKey

生产者在将消息发送给 Exchange 的时候，通常会给消息分配一个 RoutingKey，RoutingKey 定义了该消息的路由规则，但是，这个 RoutingKey 需要与 ExchangeType 和 BindingKey 共同使用才能最终决定消息投递到哪个队列中。在实际使用中，ExchangeType 与 BindingKey 通常为预先设定值。因此，消息生产者在发送消息给 Exchange 时，只需为消息设定相应的 RoutingKey，便可决定消息应该投递到哪个 Queue。通常，RabbitMQ 为 RoutingKey 设定的长度限制为 255 字节。

8. Binding

Binding 是 RabbitMQ 中将 Exchange 与 Queue 关联起来的操作，也可以看成是交换器与队列的绑定操作（如图 5-5 所示），绑定的过程需要用到消息的 RoutingKey 和绑定自身的 BindingKey。通过 Binding 操作，RabbitMQ 就知道应该如何正确地将消息路由到指定的 Queue 中。

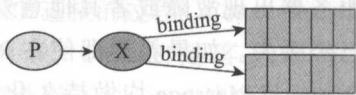


图 5-5 Exchange 与 Queue 的绑定

9. BindingKey

在绑定 Queue 到 Exchange 的过程中，通常会指定一个 BindingKey，即 BindingKey 代表了 Queue 与 Exchange 之间的对应关系。消息生产者将消息发送给 Exchange 时，通常会给消息指定一个 RoutingKey。在 Exchange 中，如果 BindingKey 与 RoutingKey 相匹配，则带有该 RoutingKey 的消息将会被路由到 BindingKey 所对应的 Queue 中，即实现了消息到特定队列的投递过程。通常情况下，消息到队列的投递过程会因 ExchangeType 的不同而有所不同。在绑定多个 Queue 到同一个 Exchange 的时候，这些 Binding 操作允许使用相同的 BindingKey。此外，并非所有 Binding 操作都需要使用 BindingKey，是否需要取决于 ExchangeType，如在 Fanout 类型的 Exchange 中，绑定操作就不会用到 BindingKey，而是将消息路由到所有绑定到该 Exchange 的 Queue 中。

10. ExchangeType

RabbitMQ 使用不同的交换器类型来将不同的消息投递到特定的队列中，不同类型的交换器使用不同的方式进行消息投递，常用的 ExchangeType 有 Fanout、Direct、Topic 和 Headers 四种。

(1) Fanout 类型

Fanout 类型的 Exchange 路由规则非常简单，它会把所有发送到该 Exchange 的消息直接投递到所有与它绑定的 Queue 中，其功能就像一个没有路由功能的 Hub，主要作用就是将多个 Queue 绑定到同一个 Exchange 中，从而实现消息生产者与队列之间一对多的对应关系。图 5-6 中，生产者 P 发送到 Exchange 的所有消息都会投递到图中的两个 Queue，并最终被两个消费者 C1 和 C2 分别读取。

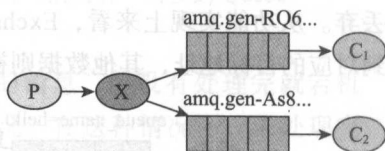


图 5-6 Fanout 类型 Exchange

在 Fanout 类型的 Exchange 中，消息到队列的投递过程并不依赖消息的 RoutingKey 和 Binding 的 BindingKey，Exchange 仅起到消息传递的作用。在 Fanout 类型中，Messages 在 RabbitMQ

Broker 内部的数据传递过程如图 5-7 所示。

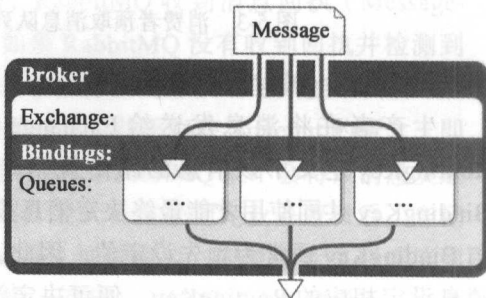


图 5-7 Fanout Exchange 消息流

(2) Direct 类型

Direct 类型的 Exchange 路由规则相对也比较简单, Direct 仅把 RoutingKey 与 Binding 的 BindingKey 匹配的消息路由到对应的 Queue 中, DirectExchange 是应用较多的 Exchange 类型之一。在图 5-8 中, 当消息生产者将 RoutingKey 为 “error” 的消息发送到 Direct 类型的 Exchange 时, 消息会被路由到 Queue1 (即队列 amqp.gen-S9b..., 这是由 RabbitMQ 自动生成的 Queue 名称) 和 Queue2 (即队列 amqp.gen-Agl...) 中。如果消息生产者以 RoutingKey 为 “info” 或 RoutingKey 为 “warning” 来发送消息, 则消息只会路由到 Queue2 (即队列 amqp.gen-Agl...) 中。

在图 5-8 中, 如果发送到 Exchange 的消息 RoutingKey 不是 “info”、“error” 和 “warning” 其中之一, 则消息不会路由到上述两个 Queue 中, 因此通常认为 DirectExchange 是一种点对点 (Point-to-Point) 的精确匹配路由模式。在 Direct 类型的 Exchange 中, Messages 在 RabbitMQ Broker 内部的数据传递过程如图 5-9 所示。

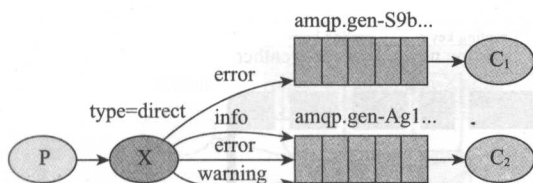


图 5-8 Direct 类型 Exchange

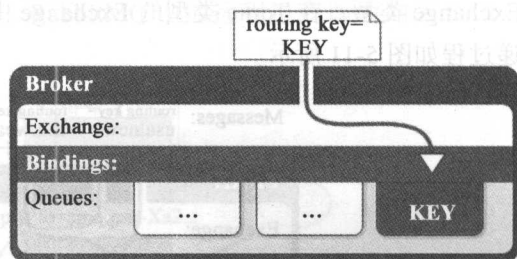


图 5-9 Direct Exchange 消息流

(3) Topic 类型

Direct 类型的 Exchange 路由规则要求完全匹配 BindingKey 与 RoutingKey, 而这种过于严格的匹配方式很难满足灵活多变的实际业务需求。Topic 类型的 Exchange 在匹配规则上进行了扩展, 其与 Direct 类型的 Exchange 相似, 也是将消息路由到 BindingKey 与 RoutingKey 相匹配的 Queue 中, 但 Topic 采用的匹配规则与 Direct 不同, Topic 采用的并非完全匹配, 而是更加灵活的模式匹配, Topic 匹配规则具有如下限定:

- ❑ RoutingKey 由点号 “.” 分隔的字符串组成 (通常将句点号 “.” 分隔开的每一段独立字符串称为一个单词), 如值为 “stock.usd.nyse” 的 RoutingKey, 其有三个单词组成, 而值为 “nyse.vmw” 的 RoutingKey, 则由两个单词组成。
- ❑ BindingKey 与 RoutingKey 一样也是由句点号 “.” 分隔的字符串。
- ❑ BindingKey 中可以包含两种特殊字符 “*” 与 “#”, 用于做模糊匹配, 其中 “*” 用于匹配一个单词, “#” 用于匹配零个或多个单词, 需要注意的是, BindingKey 中的匹配最小单位为第一条约束中定义的单词, 而不是常见的字母匹配。

图 5-10 中, RoutingKey 为 “quick.orange.rabbit” 的消息会被 Exchange 同时路由到 Q1 与 Q2 队列, 因为根据 BindingKey 的匹配规则, Q1 与 Q2 的 BindingKey 均与此 RoutingKey

匹配。RoutingKey 为 “lazy.brown.fox” 的消息只会被路由到 Q2 队列，因为 BindingKey 中的 “#” 字符匹配零个或多个单词，只有 Q2 中的 BindingKey 匹配此消息的 RoutingKey。同样，根据匹配规则，RoutingKey 为 “lazy.orange.fox” 的消息只会被路由到 Q1 队列。RoutingKey 为 “lazy.pink.rabbit” 的消息尽管与 Q2 的两个 BindingKey 都匹配，但是此消息只会投递给 Q2 一次。当消息的 RoutingKey 为 “quick.brown.fox”、“orange” 或 “quick.orange.male.rabbit” 时，由于没有任何匹配的 BindingKey 规则，这些消息将会被 Exchange 丢弃。

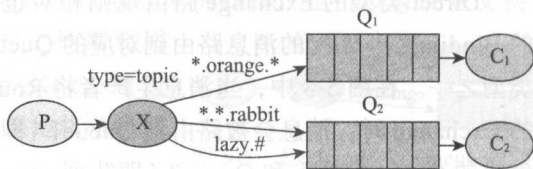


图 5-10 Topic 类型 Exchange

Topic 是一种推送 - 订阅 (Pub-Sub, Publish-Subscribe) 模式的模糊路由匹配，由于 Topic 类型的 Exchange 具有灵活自动的匹配模式，其在实际应用场景中是使用最多的 Exchange 类型。在 Topic 类型的 Exchange 中，Messages 在 RabbitMQ Broker 内部的数据传递过程如图 5-11 所示。

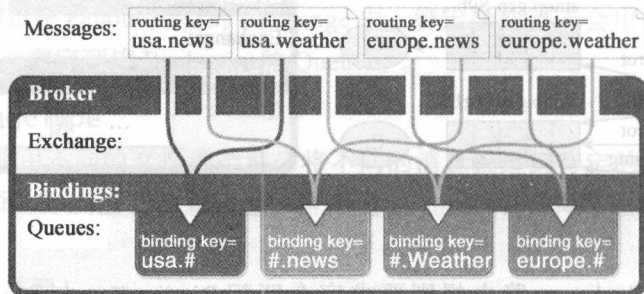


图 5-11 Topic Exchange 消息流

(4) Headers 类型

除了常见的 Fanout、Direct 和 Topic 类型之外，还有一个比较特殊的 Exchange 类型，即 Headers 类型。Headers 类型的 Exchange 不依赖于 RoutingKey 和 BindingKey 的匹配规则来路由消息，而是根据消息内容中的 Headers 属性来进行 Queue 选择。在 Headers 类型的 Exchange 中，在绑定 Queue 与 Exchange 时通常会设定一组 Key-Value 键值对，而当消息发送到 Exchange 时，RabbitMQ 会提取此消息的 Headers 值（其值也是一个 Key-Value 键值对），并对比其中的键值对是否完全匹配 Queue 与 Exchange 绑定时设定的键值对，如果完全匹配则消息会路由到该 Queue，否则不会路由到该 Queue。相比而言，Headers 类型的 Exchange 在实际应用中并不常见。

11. Remote Procedure Call

RabbitMQ 本身是一种基于异步消息的 AMQP 实现，按照 AMQP 的消息传递实现原理，消息生产者 P 只负责将消息发送到 RabbitMQ Broker，之后是否有消费者 C 来提取

队列中的消息，提取后处理成功与否，消息生产者 P 均不会知道。而在实际的应用场景中，用户很可能需要进行某些同步处理，因此需要同步等待客户端将用户发送的消息处理完成后再进行下一步处理，而这相当于远程过程调用（RPC，Remote Procedure Call），RabbitMQ 也支持远程过程调用 RPC。RabbitMQ 中实现 RPC 的机制是，消息生产者在发送消息时，在消息的属性（AMQP 协议中定义了 14 种消息属性，这些属性会随着消息一起发送）中设置两个值，分别为 ReplyTo 和 CorrelationId。其中，ReplyTo 的值是一个 Queue 名称，用于告诉消息的消费客户端消息处理完成后将应答消息发送到指定的这个 Queue 中，CorrelationId 表示此次请求的标识号，客户端处理完成后需要将此属性一并返还，消息发送端将根据返回值中的这个 id 值来判断执行成功或失败的是已经发送出去的那条消息。消息接收客户端收到消息并处理，处理完消息后，将会生成一条应答消息到 ReplyTo 指定的 Queue，同时带上 CorrelationId 属性，消息发送端之前已订阅了 ReplyTo 指定的 Queue，因此可以从中接收客户端的应答消息，并根据其中的 CorrelationId 属性分析哪条请求已经被执行，然后根据执行结果进行后续的业务处理，RabbitMQ 中的远程过程调用如图 5-12 所示。

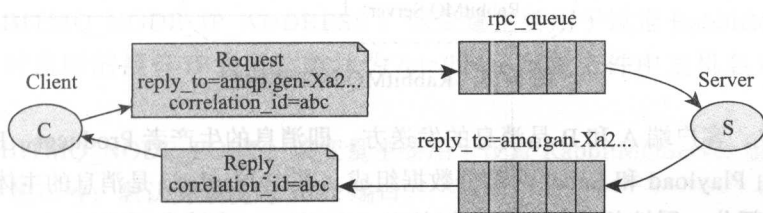


图 5-12 RabbitMQ 中 RPC 过程

5.3 RabbitMQ 工作原理

如 5.2 节所述，RabbitMQ 是 AMQP 的一个开源实现，其主要作用是在分布式集群中进行功能组件之间的异步消息传递，简单的描述就是消息生产者将带有 RoutingKey 的消息发送至交换器 Exchange，Exchange 使用 BindingKey 与 Queue 进行绑定，然后 Exchange 将 RoutingKey 与 BindingKey 进行匹配对比，并将匹配的消息投递至对应的 Queue 中。RabbitMQ 的工作流程大致如下。

- 1) 客户端连接至消息队列服务器 Broker，在 TCP 连接中建立一个虚拟 Channel。
- 2) 客户端声明一个 Exchange，并设置相关属性。
- 3) 客户端声明一个 Queue，并设置相关属性。
- 4) 客户端在 Exchange 和 Queue 之间建立好绑定关系，并设置 BindingKey。
- 5) 客户端投递带有 RoutingKey 的消息到 Exchange 中。
- 6) Exchange 接收到消息后，根据消息的 RoutingKey 和已经设置的 BindingKey，进行消息路由，将消息投递到一个或多个队列里。

RabbitMQ 的工作流程原理如图 5-13 所示，图 5-13 中的 RabbitMQ 工作原理场景主要由三个环节构成，分别是发送消息的客户端、解耦消息发送端与接收端的 RabbitMQ Server 以及消息的接收客户端。RabbitMQ Server 也称为 Broker Server，其作用主要是负责消息从 Producer 到 Consumer 的传递路径，Broker 接收从客户端发送过来的消息，然后转发给接收消息的客户端，Broker 将发送与接收客户端进行了解耦，从而实现消息发送端与接收端的异步工作。

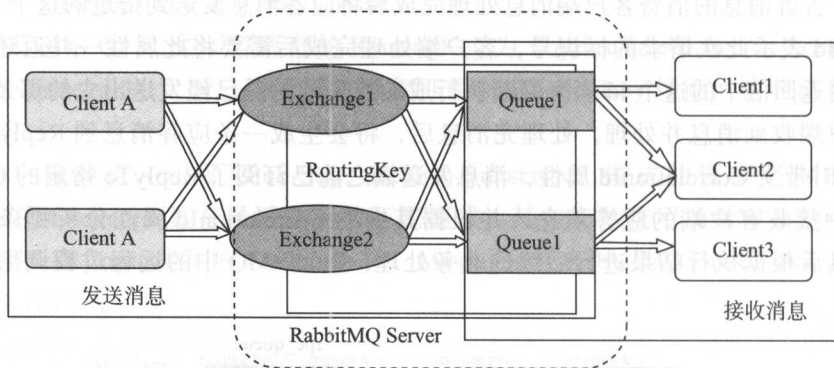


图 5-13 RabbitMQ 的工作原理

图 5-13 中，客户端 A 和 B 是消息的发送方，即消息的生产者 Producer，Producer 发出的 Messages 由 Payload 和 Label 两部分数据组成。其中 Payload 是消息的主体部分，Label 是消息的属性部分，属性部分包含了详细的 RoutingKey，当消息到达 Broker Server 后，消息会被 RabbitMQ 根据消息属性（Label 中的 RoutingKey）转发到相应的队列中。

位于原理图最右端的客户端 1、2 和 3 是消息的接收者，即消息消费者 Consumers。Broker Server 根据消息的 Label 属性和 Consumers 对队列（队列已经绑定到 Exchange 中）的订阅（Subscribe）情况，将消息的 Payload 主体转发给相应的 Consumers。Consumers 接收到的消息是没有 Label 属性的，而仅有消息的主体 Payload 部分，此外，Consumers 也不知道该消息是来自发送消息的客户端 A 还是 B。

RabbitMQ 的工作原理其实与工作中经常使用的邮件系统很类似，即图 5-13 描述的工作原理可以进行如下理解。客户 A 和 B（这里表示消息的生产者 Producer）到公共服务平台（这里代表 RabbitMQ Server）申请了一个或多个账户邮箱（Exchanges），客户 1、2 和 3 向公共服务平台申请了邮件订阅，并提供订阅接收邮箱（Queue）给服务平台。客户 A 和 B 向自己的账户邮箱发送带有标记的邮件，服务平台接收到邮件后根据邮件标记（RoutingKey）决定该邮件应该转发到哪个订阅邮箱（Queue），客户 1、2 和 3 只管到订阅邮箱去刷新读取邮件，而不会去在乎这些邮件的发送者究竟是谁。在整个过程中，RabbitMQ Server 起到了邮件调度转发的角色，而在实际的消息队列系统中，RabbitMQ 的作用与此类似，主要起到了消息接收与调度转发的作用。

5.4 RabbitMQ 基本配置

正常情况下，系统中成功安装 RabbitMQServer 程序后，用户只需启动 RabbitMQ 服务便可使其正常运行，即 RabbitMQ 使用自带的默认配置便可提供强大的异步消息传递服务。在某些情况下，用户可能希望自定义 RabbitMQ 服务，因此 RabbitMQ 提供了三种自定义配置 Broker Server 的方式，分别是环境变量配置方式、配置文件修改方式和运行时参数修改方式。

1. 环境变量配置方式

环境变量主要用于设置 IP 地址和端口，命名相应的文件并指定文件存放的位置。RabbitMQ 的环境变量具有以“RABBITMQ_”开头的前缀，在 Shell 环境变量中，RabbitMQ 的变量格式为“RABBITMQ_var_name”，其中位于前缀“RABBITMQ_”后的“var_name”表示 RabbitMQ 的环境变量名。而在 RabbitMQ 的配置文件 RabbitMQ-env.conf 中，变量格式中并无前缀“RABBITMQ_”，而是普通的变量名，如 var_name。在 RabbitMQ 的自定义配置过程中，最常使用的环境变量有以下几个。

- ❑ RABBITMQ_NODE_IP_ADDRESS：该变量主要用于设定 RabbitMQServer 服务运行时监听的接口 IP 地址，默认为 /etc/hosts 配置文件中主机名对应的接口 IP 地址。
- ❑ RABBITMQ_NODE_PORT：该变量主要用于设置 RabbitMQServer 服务运行时监听的 IP 端口号，默认为系统的 5672 端口。
- ❑ RABBITMQ_NODENAME：该变量表示 RabbitMQ 集群的节点名称，默认为 rabbit@hostname 格式，其中“hostname”为当前节点的主机名，对于 FQN 格式的主机名，如 node1.example.com，则 RabbitMQ 节点的名称为默认为 rabbit@node1。
- ❑ RABBITMQ_USE_LONGNAME：该变量的定义与 RABBITMQ_NODENAME 类似，不过此变量代表的是 RabbitMQ 的长节点名，而 RABBITMQ_NODENAME 为短节点名称。在 RabbitMQ 集群的配置中，如果此变量设置为 True，则 RabbitMQ 的节点名称将使用 FQN 全名，即 rabbit@node1.example.com。

2. 配置文件修改方式

配置文件修改方式主要用于设置 Broker Server 组件的权限、限制和集群配置，同时也用于某些插件的设置修改。RabbitMQ 主要有两个配置文件，分别为 RabbitMQ.config 和 RabbitMQ-env.conf。其中，RabbitMQ-env.conf 的位置是确定且不能自定义改变的，其位于 /etc/RabbitMQ 目录下，该文件的内容主要包括用于自定义 RabbitMQ 的环境变量，这些环境变量主要用于设置 RabbitMQ 服务运行时监听的 IP 地址和端口号、.config 配置文件的路径、Mnesia 数据库的存放路径、log 的存放路径及插件的安装配置路径。另一配置文件 RabbitMQ.config 是一个标准的 Erlang 配置文件，它必须符合 Erlang 配置文件的标准，

RabbitMQ.config 既有默认的存放目录 /etc/rabbitmq，也可以在 RabbitMQ-env.conf 文件中自定义其位置目录。

```
// RabbitMQ-env.conf 配置文件目录
[root@controller3-vm RabbitMQ]# pwd
/etc/RabbitMQ
[root@controller3-vm RabbitMQ]# ls -l
total 4
-rw-r--r-- 1 root root 32 Mar  5 21:37 RabbitMQ-env.conf
```

通常，在 RabbitMQ 软件包安装完成之后，系统中并不一定存在 RabbitMQ.config 和 RabbitMQ-env.conf 配置文件。如果 RabbitMQ-env.conf 配置文件不存在，则用户可以手动创建它，但是只能将其创建于 /etc/RabbitMQ 目录中，即 RabbitMQ-env.conf 的存放位置是固定的。如果 RabbitMQ.config 配置文件不存在，则可以手动创建它，同时可以通过设置 RABBITMQ_CONFIG_FILE 环境变量来改变 RabbitMQ.config 配置文件的存放位置，即不一定是默认的 /etc/RabbitMQ，当然还可以通过编辑 RabbitMQ-env.conf 配置文件，并设置其中的 CONFIG_FILE 变量来指定 RabbitMQ.config 配置文件的位置。

3. 运行时参数和策略

这种方式主要用于定义可以在运行时改变的集群层面的配置变量。对于 RabbitMQ 的配置而言，大多数配置都是通过环境变量定义和配置文件修改来实现，并且通过这两种方式几乎可以设置各种自定义的 RabbitMQ 运行模式，因此这种 RabbitMQ 配置定义方式很少使用。

5.5 RabbitMQ 集群基础

5.5.1 RabbitMQ 集群概述

RabbitMQ Broker 是一个或几个运行 RabbitMQ 应用的 Erlang 节点的组合，这些节点之间共享 Users、VirtualHosts、Queues、Bindings、Exchanges 和运行时参数，通常把这些运行 RabbitMQ 服务的节点组合称为 RabbitMQ 的集群。在 RabbitMQ 集群中，RabbitMQ Broker 运行所需的元数据和状态信息会自动在集群节点之间进行复制，但是需要强调的是，在普通的 RabbitMQ 集群配置中，消息队列 Queues 不会在多个节点之间复制，即集群 Queues 通常只位于集群中的某一个节点上，而其他节点虽然可以看到和访问这个节点上的消息队列，但是不会将该节点上的消息队列复制到其本地。

如果要将集群队列 Queues 进行镜像复制，则需要用到 RabbitMQ 的 HighlyAvailable 功能（RabbitMQ 集群的 HA 功能将在后文介绍）。对于普通的 RabbitMQ 集群模式，假设集群由 A 和 B 两个 RabbitMQ 节点构成，则 A、B 两个节点都有相同的集群元数据，但是只有 A（或者 B）节点持有集群消息队列，当消息进入 A 节点的 Queues 后，如果 Consumer 从 B

节点提取消息，则 RabbitMQ 会临时在 A、B 间进行消息传输，把 A 中的消息取出并经过 B 发送给 Consumer。

由于消息集中存放在 A 节点的队列中，无论 Consumer 从 A 或 B 提取消息，消息总要从 A 发出，这势必会导致 A 节点出现性能瓶颈。此外，这种普通集群模式存在的另一个问题就是当 A 节点故障后，B 节点无法提取到 A 节点中还未消费的消息实体，最终因 A 节点的单点故障而导致整个集群消息系统不可用。解决因 A 节点故障而导致消息丢失的一种办法就是将 A 节点的 Queues 持久化，尽管可以对 A 中的 Queues 做消息持久化，但在 A 故障后，也必须等待 A 节点恢复才可继续提供消息传递服务。当然，如果没有消息持久化，则即使 A 节点恢复，也无法恢复 A 中的队列消息。

在 RabbitMQ 的集群中，节点通常被划分为两类，即磁盘 (Disk/Disc) 节点和内存 (RAM) 节点。而在多数情况下，集群中的节点均默认是 Disk 节点，内存节点主要用于具有深度队列和大量 Exchanges 的集群中以提高消息传递的性能。由于内存节点将消息数据保存到内存中，因此重启内存节点会导致消息丢失。所以，如果存在内存节点，则消息队列必须做持久化，在做了持久化的消息队列中，即使在内存节点上，消息也会被保存到磁盘上，因此重启内存节点也可保证不丢失消息队列。

5.5.2 RabbitMQ 的集群配置

本节主要介绍如何实现 RabbitMQ 的集群配置，并最终实现一个三节点的 RabbitMQ 集群。三个节点的主机名分别为 rabbit1、rabbit2 和 rabbit3，现假设三个节点系统中都安装了 RabbitMQ-server 软件包，RabbitMQ 服务已经正常启动，并且 RabbitMQ 的命令行工具 RabbitMQctl 已经可以正常使用。

由于 RabbitMQ 节点之间使用 Erlang Cookie 来建立连接，因此要想让各个 RabbitMQ 节点之间彼此可以通信，则各个节点需要共享同一个 Erlang Cookie。Erlang Cookie 是在 RabbitMQ-server 启动过程中创建的一个随机字符串，在 Linux 系统中，Cookie 保存在 /var/lib/RabbitMQ/.erlang.cookie 文件中。要让 RabbitMQ 的各个节点共享同一个 Erlang Cookie，最简单的方式就是在某个节点 (rabbit1) 中启动 RabbitMQ-server，然后将 rabbit1 中的 /var/lib/RabbitMQ/.erlang.cookie 拷贝到 rabbit2 和 rabbit3 中。另外一种方式就是在启动 RabbitMQ-server 过程中或者在 RabbitMQctl 的命令行中通过参数 “-setcookie cookie_str” 的方式将特定的 Erlang Cookie 传入给当前节点的 RabbitMQ 服务。如果节点之间的 Erlang Cookie 未能正确匹配，则 RabbitMQ 的 log 中会有 “Connection attempt from disallowed node” 和 “Could not auto-Cluster” 的记录。

由于 RabbitMQ 集群是基于运行中的 RabbitMQ 节点进行配置的，因此在 Erlang Cookie 同步完成后，需要将各个节点的 RabbitMQ 服务以 detached 的方式重新启动，如下：

```

rabbit1$ RabbitMQ-server stop
rabbit2$ RabbitMQ-server stop
rabbit3$ RabbitMQ-server stop
rabbit1$ RabbitMQ-server -detached
rabbit2$ RabbitMQ-server -detached
rabbit3$ RabbitMQ-server -detached

```

待各个节点的 RabbitMQ-server 服务启动完成后，便创建了三个独立的 RabbitMQ Broker，即每个节点是一个独立的 RabbitMQ Broker，通过 RabbitMQctl 命令行工具可以验证各个节点的 Broker 运行情况，如下：

```

rabbit1$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes, [{disc, [rabbit@rabbit1]}]}, {running_nodes, [rabbit@rabbit1]}]
...done.
rabbit2$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes, [{disc, [rabbit@rabbit2]}]}, {running_nodes, [rabbit@rabbit2]}]
...done.
rabbit3$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit3 ...
[{nodes, [{disc, [rabbit@rabbit3]}]}, {running_nodes, [rabbit@rabbit3]}]
...done.

```

可以看到，每个节点上只有一个 RabbitMQ Broker 在运行，Broker 节点名分别为 rabbit@rabbit1、rabbit@rabbit2 和 rabbit@rabbit3，并且三个 Broker 节点均为磁盘类型的节点。现在，为了实现一个三节点的 RabbitMQ 集群，通常的做法是首先创建一个两节点的集群，然后再通过新增节点的形式扩展至三节点集群，即首先将 rabbit@rabbit2 加入到 rabbit@rabbit1 中，最后再将 rabbit@rabbit3 加入到 rabbit@rabbit1 和 rabbit@rabbit2 集群中。rabbit@rabbit2 加入到 rabbit@rabbit1 中的具体步骤如下：

```

rabbit2$ RabbitMQctl stop_app
Stopping node rabbit@rabbit2 ...done.
rabbit2$ RabbitMQctl join_Cluster rabbit@rabbit1
Clustering node rabbit@rabbit2 with [rabbit@rabbit1] ...done.
rabbit2$ RabbitMQctl start_app
Starting node rabbit@rabbit2 ...done.

```

现在，rabbit@rabbit2 已经成功加入到 rabbit@rabbit1 中，并且集群的名称就是 rabbit@rabbit1，可以通过 RabbitMQctl 命令行工具在两个节点上进行集群运行状态的验证：

```

// rabbit@rabbit1上检查集群运行状态
rabbit1$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes, [{disc, [rabbit@rabbit1, rabbit@rabbit2]}]},
 {running_nodes, [rabbit@rabbit2, rabbit@rabbit1]}]
...done.
// rabbit@rabbit2上检查集群运行状态

```

```
rabbit2$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes, [{disc, [rabbit@rabbit1, rabbit@rabbit2]]}],
 {running_nodes, [rabbit@rabbit1, rabbit@rabbit2]]}
...done.
```

正常情况下, 在 `rabbit@rabbit1` 和 `rabbit@rabbit2` 节点上看到的集群运行状态应该是一样的, 即集群均由两个运行的节点 `rabbit@rabbit1` 和 `rabbit@rabbit2` 组成。现在, 将第三个节点 `rabbit@rabbit3` 加入到集群中以构成三节点的 RabbitMQ 集群。此时, 选择将 `rabbit@rabbit3` 加入到 `rabbit2@rabbit2` 还是 `rabbit@rabbit1` 并没有关系, 因为选择集群中的任何一个节点, `rabbit@rabbit3` 都会加入该节点所属的集群中, 这里选择将 `rabbit@rabbit3` 加入到 `rabbit@rabbit2`, 操作步骤与 `rabbit@rabbit2` 加入 `rabbit@rabbit1` 类似:

```
rabbit3$ RabbitMQctl stop_app
Stopping node rabbit@rabbit3 ...done.
rabbit3$ RabbitMQctl join_Cluster rabbit@rabbit2
Clustering node rabbit@rabbit3 with rabbit@rabbit2 ...done.
rabbit3$ RabbitMQctl start_app
Starting node rabbit@rabbit3 ...done.
```

至此, 三节点的 RabbitMQ 集群创建完毕, 通过 RabbitMQctl 命令行工具在任何一个节点上均可验证 RabbitMQ 集群的运行状态, 并且在正常情况下, 在任一节点上所看到的集群状态应该是一致的。

```
// rabbit@rabbit1上检查集群状态
rabbit1$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes, [{disc, [rabbit@rabbit1, rabbit@rabbit2, rabbit@rabbit3]]}],
 {running_nodes, [rabbit@rabbit3, rabbit@rabbit2, rabbit@rabbit1]]}
...done.
// rabbit@rabbit2上检查集群状态
rabbit2$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes, [{disc, [rabbit@rabbit1, rabbit@rabbit2, rabbit@rabbit3]]}],
 {running_nodes, [rabbit@rabbit3, rabbit@rabbit1, rabbit@rabbit2]]}
...done.
// rabbit@rabbit3上检查集群状态
rabbit3$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit3 ...
[{nodes, [{disc, [rabbit@rabbit3, rabbit@rabbit2, rabbit@rabbit1]]}],
 {running_nodes, [rabbit@rabbit2, rabbit@rabbit1, rabbit@rabbit3]]}
...done.
```

从节点的集群状态输出中可以看到, 每个 RabbitMQBroker 节点都加入到了集群中, 并且集群由三个节点 `rabbit@rabbit3`、`rabbit@rabbit2`、`rabbit@rabbit1` 组成, 每个节点上都在运行 RabbitMQ 服务, 并且节点默认都是磁盘类型的节点。按照以上集群创建和扩展方法, 可以继续增加更多的节点到 RabbitMQ 集群中。

5.6 RabbitMQ 集群管理

5.6.1 RabbitMQ 集群节点启停

在一个正在运行的 RabbitMQ 集群中，可以将任何一个或多个节点停止，并且集群中剩下的节点将会正常运行而不会受到某个节点停止的影响。而当停止的节点重新启动后，该节点又会自动加入集群，集群状态自动恢复正常。现在，依次停止集群中的 `rabbit@rabbit1` 和 `rabbit@rabbit3` 节点，并在每停止一个节点后，先观察集群的运行状态。

```
//停止rabbit@rabbit1节点
rabbit1$ RabbitMQctl stop
Stopping and halting node rabbit@rabbit1 ...done.
//在rabbit@rabbit2上观察rabbit@rabbit1停止后集群的运行状态
rabbit2$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2,rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit3,rabbit@rabbit2]}]}
...done.
//在rabbit@rabbit3上观察rabbit@rabbit1停止后集群的运行状态
rabbit3$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit3 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2,rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit2,rabbit@rabbit3]}]}
...done.
//停止rabbit@rabbit3节点
rabbit3$ RabbitMQctl stop
Stopping and halting node rabbit@rabbit3 ...done.
//在rabbit@rabbit2上观察集群运行状态
rabbit2$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2,rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit2]}]}
...done.
```

可以看到，当节点 `rabbit@rabbit1` 停止后，集群中的节点依然是 `rabbit@rabbit1`、`rabbit@rabbit2` 和 `rabbit@rabbit3`，但是运行中的节点只有 `rabbit@rabbit2` 和 `rabbit@rabbit3`。当 `rabbit@rabbit3` 停止后，集群中的节点成员仍然没有改变，但是运行的节点只有 `rabbit@rabbit2`，即此时只有 `rabbit@rabbit2` 能够对外提供消息服务。现在，将之前已经停止的 `rabbit@rabbit1` 和 `rabbit@rabbit3` 节点重新启动，然后分别观察节点启动后集群的运行状态。

```
//以detached方式启动rabbit@rabbit1节点
rabbit1$ RabbitMQ-server -detached
//在rabbit@rabbit1上观察集群运行状态
rabbit1$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2,rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit2,rabbit@rabbit1]}]}
...done.
```

```

//在rabbit@rabbit2上观察集群运行状态
rabbit2$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes, [{disc, [rabbit@rabbit1, rabbit@rabbit2, rabbit@rabbit3]}]},
 {running_nodes, [rabbit@rabbit1, rabbit@rabbit2]}]
...done.
//以detached方式启动rabbit@rabbit3节点
rabbit3$ RabbitMQ-server -detached
//在rabbit@rabbit1上观察集群运行状态
rabbit1$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes, [{disc, [rabbit@rabbit1, rabbit@rabbit2, rabbit@rabbit3]}]},
 {running_nodes, [rabbit@rabbit2, rabbit@rabbit1, rabbit@rabbit3]}]
...done.
//在rabbit@rabbit2上观察集群运行状态
rabbit2$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes, [{disc, [rabbit@rabbit1, rabbit@rabbit2, rabbit@rabbit3]}]},
 {running_nodes, [rabbit@rabbit1, rabbit@rabbit2, rabbit@rabbit3]}]
...done.
//在rabbit@rabbit3上观察集群运行状态
rabbit3$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit3 ...
[{nodes, [{disc, [rabbit@rabbit1, rabbit@rabbit2, rabbit@rabbit3]}]},
 {running_nodes, [rabbit@rabbit2, rabbit@rabbit1, rabbit@rabbit3]}]
...done.

```

可以看到，当重新启动 `rabbit@rabbit1` 节点后，`rabbit@rabbit1` 会自动加入集群运行，重新启动 `rabbit@rabbit3` 节点后，三节点集群状态自动恢复如初，即集群中三个集群成员都在运行，并可同时对外提供消息服务。在 RabbitMQ 全部集群节点均被停止并需重新重启时，有两点需要特别指出，如下所示。

- ❑ 如果整个 RabbitMQ 集群中的节点都停止，则重启时应该根据节点停止顺序的逆序重新启动节点。如果首先启动的不是最后停止的节点，则启动过程中会给出 30s 的等待时间，以等待最后停止的节点变为运行状态，如果在 30s 内最后停止节点仍然未能激活，则节点启动过程就以失败告终。而如果最后停止的节点无法启动，则可以使用 `forget_Cluster_node` 命令将该节点从集群中移除，`forget_Cluster_node` 命令的具体使用方式可以参考 `RabbitMQctl` 命令行工具的帮助页面。
- ❑ 如果整个集群节点被同时停止或者发生了掉电等意外情况导致全部 RabbitMQ 节点同时关闭，则集群中每个节点都会认为自己不是集群中最后停止的节点，而会认为其他节点将在自己后面停止。如果出现这种情况，则可使用 `RabbitMQctl` 的 `force_boot` 命令来重新启动节点。

5.6.2 RabbitMQ 的集群节点移除

在正常运行的 RabbitMQ 集群中，当一个节点再也无须加入 RabbitMQ 集群并作为其

成员节点运行的时候,就需要将此节点从集群中移除。要移除集群节点,首先需要停止该节点的 RabbitMQ 应用,然后重置节点,最后重启该节点的 RabbitMQ 应用。现在,假设需要将 rabbit@rabbit3 节点从集群中移除并使其成为独立的运行 RabbitMQ Broker,操作步骤如下:

```
//停止
rabbit3$ RabbitMQctl stop_app
Stopping node rabbit@rabbit3 ...done.
//重置
rabbit3$ RabbitMQctl reset
Resetting node rabbit@rabbit3 ...done.
//重启
rabbit3$ RabbitMQctl start_app
Starting node rabbit@rabbit3 ...done.
```

现在 rabbit@rabbit3 节点已经从集群中移除,并且已成为一个独立运行的 RabbitMQ Broker,可以通过 Cluster_status 命令来验证集群节点情况,如下:

```
rabbit1$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2]}]},
 {running_nodes,[rabbit@rabbit2,rabbit@rabbit1]}}]
...done.
rabbit2$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2]}]},
 {running_nodes,[rabbit@rabbit1,rabbit@rabbit2]}}]
...done.
rabbit3$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit3 ...
[{nodes,[{disc,[rabbit@rabbit3]}]}, {running_nodes,[rabbit@rabbit3]}]
...done.
```

可见,在移除 rabbit@rabbit3 后,集群中只有 rabbit@rabbit1 和 rabbit@rabbit2 两个节点,同时 rabbit@rabbit3 节点中只有一个 RabbitMQ Broker 在运行。如果集群中某个节点已经失去了响应,且不能通过正常方式将其移除,则可以在本地节点通过 forget_Cluster_node 命令以远程移除的方式来将此节点从集群中移除。现在,将 rabbit@rabbit1 节点以 forget_Cluster_node 命令方式使其从集群中移除,操作步骤如下:

```
rabbit1$ RabbitMQctl stop_app
Stopping node rabbit@rabbit1 ...done.
rabbit2$ RabbitMQctl forget_Cluster_node rabbit@rabbit1
Removing node rabbit@rabbit1 from Cluster ...
...done.
```

rabbit@rabbit1 在本地节点以远程方式从集群中移除后,其仍然会认为自己还属于集群节点,因此在启动本地 rabbit@rabbit1 节点的 RabbitMQ 应用时会报错,在重启之前将其重置即可解决,重置过程如下:

```

rabbit1$ RabbitMQctl start_app
Starting node rabbit@rabbit1 ...
Error: inconsistent_Cluster: Node rabbit@rabbit1 thinks it's Clustered with node
rabbit@rabbit2, but rabbit@rabbit2 disagrees
rabbit1$ RabbitMQctl reset
Resetting node rabbit@rabbit1 ...done.
rabbit1$ RabbitMQctl start_app
Starting node rabbit@mcnulty ...
...done.

```

至此, `rabbit@rabbit1` 和 `rabbit@rabbit3` 均从集群脱离, 通过 `Cluster_status` 命令可以看到三个节点又都回到各自最初的独立节点状态, 即又恢复为三个独立运行的 RabbitMQ Broker。

```

rabbit1$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes, [{disc, [rabbit@rabbit1]}]}, {running_nodes, [rabbit@rabbit1]}]
...done.
rabbit2$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes, [{disc, [rabbit@rabbit2]}]}, {running_nodes, [rabbit@rabbit2]}]
...done.
rabbit3$ RabbitMQctl Cluster_status
Cluster status of node rabbit@rabbit3 ...
[{nodes, [{disc, [rabbit@rabbit3]}]}, {running_nodes, [rabbit@rabbit3]}]
...done.

```

尽管 `rabbit@rabbit1` 和 `rabbit@rabbit3` 都已脱离集群, 但是 `rabbit@rabbit2` 仍然保存有集群的状态信息, 即 `rabbit@rabbit2` 仍然还是集群成员, 只不过集群有且只有 `rabbit@rabbit2` 一个成员。而 `rabbit@rabbit1` 和 `rabbit@rabbit3` 已经重新初始化为独立的 RabbitMQ Broker, 因此这两个节点不会保存任何集群信息, 而如果要 `rabbit@rabbit2` 也重新初始化为独立的 RabbitMQ Broker, 只需执行以下几个步骤即可:

```

rabbit2$ RabbitMQctl stop_app
Stopping node rabbit@rabbit2 ...done.
rabbit2$ RabbitMQctl reset
Resetting node rabbit@rabbit2 ...done.
rabbit2$ RabbitMQctl start_app
Starting node rabbit@rabbit2 ...done.

```

5.7 RabbitMQ 的集群队列镜像

在默认的 RabbitMQ 集群中, 消息队列不会在集群节点之间进行复制备份, 而仅位于集群中的某个节点上, 通常该节点是最初声明 Queues 的节点。与 Queues 不同, Exchanges 和 Bindings 信息会复制到集群中的每个节点上, 因此在默认的 RabbitMQ 集群配置中, 尽管集群的 Exchanges 和 Bindings 信息能够避免单点故障, 但是由于 Queues 及其保存的

Messages 是中心化单点存放的，所以集群中的消息队列仍然具有单点故障而无法实现彻底的高可用，如果拥有 Queues 的节点发生故障，则虽然整个 RabbitMQ 集群可以继续提供消息服务，但是之前位于 Queues 中还未被取走的消息将会丢失或者暂时不可用（消息持久化的情况下）。

为了避免消息队列的单点故障，通常的做法是将队列在节点之间进行镜像复制。在队列镜像模式下，每个镜像的队列由一个 Master 和一个或多个 Slave 提供。如果当前的 Master 故障，则成为 Slave 角色时间最长的节点会被 Promote 为新的 Master 节点。同时，消息生产者投递到 Queues 中的消息会被复制到所有 Slave 节点上，并且不论消费者 Consumers 连接到哪个集群节点，其最终都是到 Master 节点的 Queues 中提取消息。又因为 Master 需要将消息复制到多个 Slaves 节点，所以队列镜像模式虽然增加了 RabbitMQ 集群的高可用性，但是并没有将集群的消息服务负载分散到每个集群节点中。

实现 RabbitMQ 集群队列镜像的最主要方式就是 RabbitMQ 提供的 Policy 功能，Policy 功能可以在集群运行的任何时候使用，即可以动态地将一个未镜像的集群消息队列改变为镜像队列。因此，创建镜像队列最简单有效的方式就是先创建一个非镜像的队列，然后通过 Policy 设置成为镜像队列。值得注意的是，非镜像队列与没有任何 Slave 节点的镜像队列之间是有明显区别的，非镜像队列因为没有额外的镜像操作，所以其运行效率相对要高很多。在 RabbitMQ 集群的镜像队列设置中，用户可以选择性地对某些队列进行镜像，而其他队列可以不用镜像。这种灵活配置最大的好处是，相对全镜像，部分镜像可以提供更好的系统性能。这种灵活的镜像方式是通过配置 Policy 的模式匹配来实现的，通过模式匹配，被匹配到的队列将会自动镜像，而被过滤掉的队列则会保持原状态。RabbitMQ 设置 Policy 时，最关键的两个参数分别是 ha-mode 和 ha-params，ha-params 参数的值根据 ha-mode 的取舍不同而不同。ha-mode 和 ha-params 参数可能的值及其解释如表 5-1 所示。

表 5-1 RabbitMQ 队列镜像 policy 的参数说明

ha-mode	ha-params	说 明
all	NULL	将队列镜像到全部集群节点上
exactly	count	将队列镜像到 count 个集群节点上；如果实际节点数目小于 count 值，则队列被镜像到全部节点上；如果实际节点数目大于 count，则拥有队列的节点故障后，队列会被重新镜像到其他节点上
nodes	node names	将队列镜像到 node names 指定的特定节点上

为了演示 RabbitMQ 集群队列镜像 Policy 的使用方式，下列代码段使用了 ha_mode 和 ha_params 的不同组合来进行队列镜像的设置，在使用过程中可以选择适合自己的方式进行 RabbitMQ 的队列镜像设置。

```
//将名称全部以“.ha”开头的队列镜像到所有集群节点中
RabbitMQctl set_policy ha-all "^ha\." '{"ha-mode":"all"}'
//将名称全部以“.ha”开头的队列镜像到集群中任意两个节点；
```

```
RabbitMQctl set_policy ha-two "^ha\." '{"ha-mode":"exactly","ha-params":2,"ha-sync-mode":"automatic"}'
```

//将名称全部以“nodes.”开头的队列镜像到集群中特定节点上:

```
RabbitMQctl set_policy ha-nodes "^nodes\." '{"ha-mode":"nodes","ha-params":["rabbit@nodeA", "rabbit@nodeB"]}'
```

队列镜像中的一个特殊情况是独占队列，独占队列在声明队列的连接终止后会被 RabbitMQ 自动清除，就如程序段中的一个临时变量。因此，对独占队列做镜像或者持久化并没有太大意义，因为一旦拥有这些独占队列的主机节点意外故障后，该主机所监听连接就会被强制关闭，而不管队列是否做持久化或者镜像，在其上的 RabbitMQ 队列均会被清除。所以，即使队列名称匹配镜像策略的模式，独占队列也不会被 RabbitMQ 的镜像策略进行镜像。同时，即使对独占队列进行持久化声明，其也不会被持久化。

此外，在 RabbitMQ 集群中，设置了镜像策略节点上的队列并非任何时候都是彼此同步的。如果一个节点新加入集群，并且集群 Policy 将其设为队列镜像的节点，则队列会将此节点当做一个新的 Slave 节点。但是，此时的新 Slave 节点上并没有任何的队列，或者说此时新 Slave 节点上的队列是空的。正常情况下，新加入的 Slave 节点只会接收晚于其加入镜像队列时间点新增的消息，并最终随着时间推移，新加入的 Slave 节点中的队列消息会逐步与原集群队列尾部的消息同步，并且随着原有队列头部消息被不断消耗，新 Slave 节点队列中不同步的消息数目会不断减少并最终达到完全同步。

因此，基于 RabbitMQ 的这种队列同步模式，默认设置下，新加入的 Slave 节点对其加入前已经存在的集群消息并不能形成任何的冗余和高可用，除非对原有集群队列执行显式的同步操作。由于在大型的集群队列系统中，显式同步队列会使得队列响应变得迟钝，因此，在这种情况下，推荐的方式就是让活动队列消息不断被消耗并最终实现自动同步，同时仅对不活动的队列执行显式的镜像同步操作。

从 RabbitMQ3.6 开始，RabbitMQ 新增了 ha-sync-batch-size 参数，使得显式的队列同步在速度上有了极大提升，显示队列同步可以有两种实现方式，即手工同步和自动同步。如果队列被配置为自动同步，则不论新的 Slave 节点何时加入集群，集群中的队列都会被自动同步到新增的 Slave 节点。但是，同步过程中的消息队列相应地会变迟钝，这种响应变慢的过程会一直持续到队列同步完成为止。队列自动同步的设置很简单，只需在 Policy 的镜像设置中指定 ha-sync-mode 参数，使其值为 automatic 即可。ha-sync-mode 参数允许的值是 manual 和 automatic，如果不显式指定 automatic，则其值默认为 manual，下面的 Policy 策略将队列配置为自动镜像模式。

```
//将全部以"ha"开头的队列镜像到所有节点并且自动镜像的策略:
```

```
RabbitMQctl set_policy ha-two "^ha\." '{"ha-mode":"all","ha-sync-mode":"automatic"}'
```

默认情况下，队列在镜像时对消息进行逐条同步，而在 RabbitMQ3.6 之后，新增了批量同步参数 ha-sync-batch-size，用户通过设置该参数的值，即可实现批量同步消息。如果未显式设置 ha-sync-batch-size 的值，则其值默认为 1，即不进行批量同步。在队列同步过

程中, 通过 RabbitMQctl 命令行工具即可查看哪些队列已经实现了同步, 例如在 Pacemaker 集群管理器控制下的 RabbitMQ 集群环境中, 通过如下命令可以看到队列的镜像情况:

```
[root@controller1-vm ~]# RabbitMQctl list_Queue name slave_pids synchronised_
slave_pids|more
Listing Queues ...
q-agent-notifier-tunnel-update_fanout_bela1840f2d242b9a4a8c80d746c4c5f [<rabbit@
controller1-vm.2.1000.0>, <rabbit@controller2-vm.3.798.0>]      [<rabbit@
controller2-vm.3.798.0>, <rabbit@controller1-vm.2.1000.0>]
metering.controller2-vm [<rabbit@controller2-vm.3.1668.0>, <rabbit@controller3-
vm.3.26612.3>]      [<rabbit@controller2-vm.3.1668.0>, <rabbit@controller3-
vm.3.26612.3>]
heat-engine-listener.76666c92-e20a-402e-bf66-a5de859e3abb      [<rabbit@
controller2-vm.3.2094.0>, <rabbit@controller3-vm.3.26993.3>]      [<rabbit@
controller2-vm.3.2094.0>, <rabbit@controller3-vm.3.26993.3>]
metering.controller3-vm [<rabbit@controller2-vm.3.1691.0>, <rabbit@controller3-
vm.3.26649.3>]      [<rabbit@controller2-vm.3.1691.0>, <rabbit@controller3-
vm.3.26649.3>]
conductor.controller2-vm      [<rabbit@controller2-vm.3.1543.0>, <rabbit@controller3-
vm.3.26510.3>]      [<rabbit@controller2-vm.3.1543.0>, <rabbit@controller3-
vm.3.26510.3>]
engine_fanout_ad38f273ae8c4ceba1eeefa344db5a1dc [<rabbit@controller2-vm.3.2017.0>,
<rabbit@controller3-vm.3.26953.3>]      [<rabbit@controller2-vm.3.2017.0>,
<rabbit@controller3-vm.3.26953.3>]
q-agent-notifier-port-update      [<rabbit@controller1-vm.2.1004.0>, <rabbit@
controller2-vm.3.802.0>]      [<rabbit@controller2-vm.3.802.0>, <rabbit@
controller1-vm.2.1004.0>]
```

在 list_Queue 命令的输出结果中, 第一列是 RabbitMQ 的队列名称, 第二列是由两个节点组成的字符串, 表示集群中的全部 Slave 节点, 第三列也是由两个节点组成的字符串, 表示第一列给出的队列在这两个 Slave 节点中均实现了镜像同步。上述输出结果源自三节点的 RabbitMQ 集群, 三个节点的主机名分别为 controller1-vm、controller2-vm 和 controller3-vm, 其中 controller3-vm 为 RabbitMQ Broker 的 Master 节点, 其余两个为 Slave 节点。以队列 q-agent-notifier-port-update 为例, 该队列在两个 Slave 节点 (即 controller2-vm 和 controller1-vm) 中都实现了镜像同步 (第三列的输出中包含了这两个节点)。在 Pacemaker 的管理下, RabbitMQ 服务在 Pacemaker 资源中的状态应该如下所示, 即一个 Master 节点和两个 Slave 节点。

```
Master/Slave Set: RabbitMQ-Cluster-master [RabbitMQ-Cluster]
Masters: [ controller3-vm ]
Slaves: [ controller1-vm controller2-vm ]
```

除了让 RabbitMQ 对队列进行自动同步, 用户还可以实现手工同步, 手工同步之前需要通过 list_Queue 命令找到需要同步的队列, 然后即可对指定的队列执行镜像同步或者镜像取消操作, 手工对特定队列执行镜像同步和镜像取消的命令语法如下:


```
//手动镜像同步队列
RabbitMQctl sync_Queue Queue_name
//手动取消队列镜像
RabbitMQctl cancel_sync_Queue Queue_name
```

5.8 基于 Pacemaker 的高可用 RabbitMQ 集群

5.8.1 Active/Passive 模式的 RabbitMQ 集群

RabbitMQ 可以工作在 Active/Passive 或者 Active/Active 模式的集群中。当 RabbitMQ 集群运行在 Active/Passive 模式时, 如果 Active 节点故障, 则正常运行时由 Active 节点持久化写入磁盘中的消息队列可以被 Passive 节点恢复。当然, 在 Active/Passive 模式下, 如果 Active 节点的消息队列没有进行持久化操作, 则 Active 节点故障后位于其上的消息就会丢失, 尽管 Passive 节点可以重新提供消息服务, 但是之前未被取走的消息却已无法恢复, 因此 Active/Passive 模式下的消息队列必须进行持久化操作。Active/Passive 模式的另一可能的问题是, 当 Active 节点故障, 并需要提升 Passive 节点为 Active 节点以恢复和接管消息时, RabbitMQ 集群的消息服务可能会出现短暂中断。

在 Active/Passive 模式的 RabbitMQ 集群中, Pacemaker 通常被用作集群资源管理和监控模块, 同时使用 DRBD 来提供存放持久化消息的共享存储, 当然, 如果有 NAS 或者 SAN 存储环境, 也可用其来替换 DRDB。由于 DRDB 共享存储同一时刻只允许一个节点对存储设备进行写操作, 因此不会存在多个节点同时写数据而导致文件锁争抢和数据不一致的情况发生。在 RabbitMQ 高可用集群中使用共享存储的目的, 是将 Active 节点的消息队列持久化存储到共享存储中, 当 Active 节点故障之后, 该节点对共享存储的写锁定将会被释放, 这时 Passive 节点被提升为 Active 节点, 同时获得了对共享存储的读写权限, 从而可以将存储在共享存储中的消息队列进行恢复, 并继续提供消息服务并往共享存储中写入消息。RabbitMQ 集群的 Active/Passive 高可用模式的架构设计如图 5-14 所示。

尽管 Active/Passive 高可用模式在一定程度上也可以确保 RabbitMQ 集群的高可用性, 但是 Active/Passive 模式下的 RabbitMQ 集群也存在一些问题, 如当 Active 节点故障后, 在 Passive 节点上的 RabbitMQ 服务可能会花费很长时间才能成功启动或者根本无法启动。这种情况会导致集群消息临时不可用, 直到 Active 节点重新启动并恢复消息服务为止, 因此, 在生产环境下并不推荐 RabbitMQ 集群部署在 Active/Passive 模式。

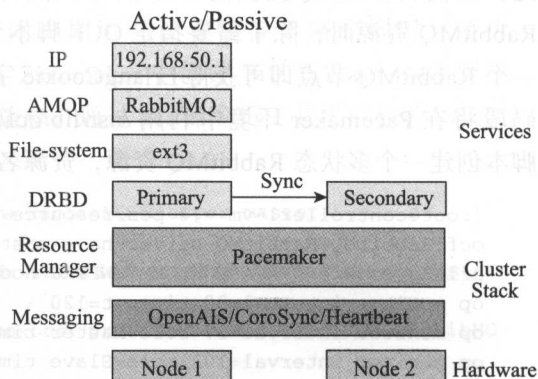


图 5-14 Active/Passive 模式的 RabbitMQ 高可用集群

5.8.2 Active/Active 模式的 RabbitMQ 集群

为了解决 Active/Passive 模式下 RabbitMQ 集群的高可用问题，RabbitMQ 社区又开发和设计了 RabbitMQ 服务的 Active/Active 集群模式。Active/Active 高可用模式的本质是将 RabbitMQ 集群中的队列在集群节点上实现彼此镜像，在 Active/Active 模式下，集群中的某一 RabbitMQ 节点故障后，RabbitMQ 服务会自动切换到其他节点，并使用该节点上的镜像队列继续提供消息服务，从而实现消息系统的服务高可用性。在部署 OpenStack 的高可用集群中，推荐部署的 RabbitMQ 高可用模式便是 Active/Active 高可用集群模式。

Active/Active 集群模式通常使用 Pacemaker 作为集群资源管理器，同时使用资源代理（RA，Resource Agent）来配置高可用 RabbitMQ 集群。由于开源的原因，RabbitMQ 有不同版本的安装包，推荐使用 Redhat 或者 RabbitMQ 的官方安装包，这里以官方 RabbitMQ3.6 安装包为例，当 RabbitMQ 安装完成后，在操作系统的 /usr/lib/ocf/resource.d 目录下将会生成 RabbitMQ 目录，该目录下存在三个 OCF 脚本，三个脚本分别针对不同的 RabbitMQ 配置功能。

- ❑ RabbitMQ-server：这是一个普通的类似 /etc/init.d 的服务启动脚本，主要用于启动 RabbitMQ server 服务。
- ❑ RabbitMQ-server-ha：RabbitMQ 集群的 Active/Active 高可用模式配置脚本，Pacemaker 将使用此脚本进行 RabbitMQ 服务高可用性的配置和管理。
- ❑ set_RabbitMQ_policy.sh：设置集群镜像队列的策略脚本，在使用 RabbitMQ-server-ha 作为 OCF 脚本配置了高可用集群后，Pacemaker 在启动 RabbitMQ 服务时会自动调用该脚本进行镜像设置，同时用户也可以自定义该脚本，使得 RabbitMQ 服务在启动后立刻进行某些用户指定的配置。

在 Pacemaker 环境下，配置 Active/Active RabbitMQ 高可用集群，只需使用 RabbitMQ 安装程序提供的 OCF 脚本创建一个多状态资源（Multi-State Resource）即可，需要注意的是，当前 RabbitMQ 提供的 OCF 脚本只支持 Disc 类型的节点。在 Pacemaker 环境中创建 RabbitMQ 资源时，除了需要指定 OCF 脚本外，还需提供 ErlangCookie 字符串（任意启动一个 RabbitMQ 节点即可获得 ErlangCookie 字符串）和 RabbitMQ 需要监听的端口。如下代码段将在 Pacemaker 环境中利用 /usr/lib/ocf/resource.d/RabbitMQ/RabbitMQ-server-ha OCF 脚本创建一个多状态 RabbitMQ 资源，资源名称为 RabbitMQ-Cluster。

```
[root@controller1-vm ~]# pcs resource create RabbitMQ-Cluster \
ocf:RabbitMQ:RabbitMQ-server-ha --master \
erlang_cookie=DPMDALGUKEOMPTHWPYKC node_port=5672 \
op monitor interval=30 timeout=120 \
op monitor interval=27 role=Master timeout=120 \
op monitor interval=103 role=Slave timeout=120 OCF_CHECK_LEVEL=30 \
op start interval=0 timeout=120 \
op stop interval=0 timeout=120 \
op promote interval=0 timeout=60 \
```

```
op demote interval=0 timeout=60 \
op notify interval=0 timeout=60 \
meta notify=true ordered=false interleave=false master-max=1 \
master-node-max=1
```

在 Pacemaker 中创建完成 RabbitMQ 资源后, RabbitMQ 的 RA 在启动资源的过程中会自动调用 RabbitMQ 安装程序提供的镜像策略脚本进行队列的镜像设置, 该策略脚本便是 /usr/lib/ocf/resource.d/RabbitMQ/set_RabbitMQ_policy.sh。默认情况下, 该脚本只进行镜像策略设置, 但是用户可以根据实际情况对该脚本进行自定义处理, 比如在启动过程中创建 RabbitMQ 用户并对其进行授权等操作, 默认的 set_RabbitMQ_policy.sh 的内容如下:

```
$ cat /usr/lib/ocf/resource.d/RabbitMQ/set_RabbitMQ_policy.sh
# This script is called by RabbitMQ-server-ha.ocf during RabbitMQ
# Cluster start up. It is a convenient place to set your Cluster
# policy here, for example:
# ${OCF_RESKEY_ctl} set_policy ha-all "." '{"ha-mode":"all",
"ha-sync-mode":"automatic"}' --apply-to all --priority 0
```

通常在使用之前, 需要对该脚本进行修改以设置 RabbitMQ 队列的镜像策略, 在 Openstack 高可用集群部署中, Openstack 的各个组件将会使用 RabbitMQ 用户来登录和访问 RabbitMQ, 因此需要为 Openstack 的各个组件创建 RabbitMQ 用户, 并对其进行授权操作, 这一步骤也可以在该脚本中完成。因此, 在实际部署中 set_RabbitMQ_policy.sh 脚本的内容如下所示。

```
$ cat /usr/lib/ocf/resource.d/RabbitMQ/set_RabbitMQ_policy.sh
# This script is called by RabbitMQ-server-ha.ocf during RabbitMQ
# Cluster start up. It is a convenient place to set your Cluster
# policy here.
${OCF_RESKEY_ctl} set_policy ha-all "." '{"ha-mode":"all", \
"ha-sync-mode":"automatic"}' --apply-to all --priority 0
${OCF_RESKEY_ctl} add_user openstack openstack
${OCF_RESKEY_ctl} set_permissions openstack ".*" ".*" ".*"
```

通过自定义 set_RabbitMQ_policy.sh 脚本, 用户即可在设置镜像策略的同时, 还为 RabbitMQ 集群创建用户并对用户进行授权。在 Pacemaker 中创建多状态 RabbitMQ 资源后, Pacemaker 的 CIB 中便保存了该资源的配置信息, 通过 pcs 命令行工具可以校验和核对 RabbitMQ-Cluster 资源的配置情况:

```
[root@controller1-vm ~]# pcs resource show RabbitMQ-Cluster-master
Master: RabbitMQ-Cluster-master
Meta Attrs: notify=true ordered=false interleave=false master-max=1 master-
node-max=1
Resource: RabbitMQ-Cluster (class=ocf provider=RabbitMQ type=RabbitMQ-
server-ha)
Attributes: erlang_cookie=DPMDALGUKEOMPTHWPYKC node_port=5672
Operations: monitor interval=30 timeout=120 (RabbitMQ-Cluster-
monitor-interval-30)
```

```
monitor interval=27 role=Master timeout=120 (RabbitMQ-Cluster-monitor-interval-27-
role-Master)
monitor interval=103 role=Slave timeout=120 OCF_CHECK_LEVEL=30 (RabbitMQ-Cluster-
monitor-interval-103-role-Slave)
start interval=0 timeout=120 (RabbitMQ-Cluster-start-interval-0)
stop interval=0 timeout=120 (RabbitMQ-Cluster-stop-interval-0)
promote interval=0 timeout=60 (RabbitMQ-Cluster-promote-interval-0)
demote interval=0 timeout=60 (RabbitMQ-Cluster-demote-interval-0)
notify interval=0 timeout=60 (RabbitMQ-Cluster-notify-interval-0)
```

由于配置的是多状态资源，当 Pacemaker 资源启动完成后，Pacemaker 集群中应该只有一个 RabbitMQ 节点作为 Master 提供消息服务，其他 RabbitMQ 节点都处于 Slave 状态，RabbitMQ 的资源状态如下：

```
[root@controller1-vm ~]# pcs status|grep RabbitMQ-Cluster-master -A2
Master/Slave Set: RabbitMQ-Cluster-master [RabbitMQ-Cluster]
Masters: [ controller1-vm ]
Slaves: [ controller2-vm controller3-vm ]
```

另外，需要指出的是，尽管在 Pacemaker 集群资源中看到的 RabbitMQ 集群由一个 Master 和两个 Slave 节点构成，但是运行中的 RabbitMQ 集群却是 Active-Active 高可用模式的，通过 RabbitMQctl 命令行工具的策略查看命令 `list_policies`，可以看到 RabbitMQ 中的队列镜像模式：

```
[root@controller1-vm ~]# RabbitMQctl list_policies
Listing policies .../
ha-all all . {"ha-mode":"all","ha-sync-mode":"automatic"} 0
```

从 RabbitMQ 集群的 Policy 中可以看到，当前 RabbitMQ 集群对全部队列都做了镜像，并且使用的是自动镜像模式，而且设定的是镜像到全部 Slave 节点。通过 RabbitMQctl 命令行工具的队列查看命令 `list_Queue`s 可以验证当前 RabbitMQ 集群中的队列镜像情况：

```
//在slave节点controller1-vm上查看队列镜像情况
[root@controller1-vm ~]# RabbitMQctl list_Queue s name slave_pids synchronised_
slave_pids|more
Listing Queue s ...
q-agent-notifier-tunnel-update_fanout_bela1840f2d242b9a4a8c80d746c4c5
f [<rabbit@controller2-vm.2.1000.0>, <rabbit@controller3-vm.3.798.0>]
[<rabbit@controller2-vm.3.798.0>, <rabbit@controller3-vm.2.1000.0>]
metering.controller2-vm [<rabbit@controller2-vm.3.1668.0>, <rabbit@controller3-
vm.3.26612.3>] [<rabbit@controller2-vm.3.1668.0>, <rabbit@controller3-
vm.3.26612.3>]
heat-engine-listener.76666c92-e20a-402e-bf66-a5de859e3abb [<rabbit@
controller2-vm.3.2094.0>, <rabbit@controller3-vm.3.26993.3>] [<rabbit@
controller2-vm.3.2094.0>, <rabbit@controller3-vm.3.26993.3>]
//在slave节点controller2-vm上查看队列镜像情况
[root@controller2-vm ~]# RabbitMQctl list_Queue s name slave_pids synchronised_
slave_pids|more
```

Listing Queues ...

```
q-agent-notifier-tunnel-update_fanout_bela1840f2d242b9a4a8c80d746c4c5f [<rabbit@
controller2-vm.2.1000.0>, <rabbit@controller3-vm.3.798.0>] [<rabbit@
controller2-vm.3.798.0>, <rabbit@controller3-vm.2.1000.0>]
metering.controller2-vm [<rabbit@controller2-vm.3.1668.0>, <rabbit@controller3-
vm.3.26612.3>] [<rabbit@controller2-vm.3.1668.0>, <rabbit@controller3-
vm.3.26612.3>]
heat-engine-listener.76666c92-e20a-402e-bf66-a5de859e3abb [<rabbit@
controller2-vm.3.2094.0>, <rabbit@controller3-vm.3.26993.3>] [<rabbit@
controller2-vm.3.2094.0>, <rabbit@controller3-vm.3.26993.3>]
```

//在master节点controller3-vm上查看队列镜像情况

```
[root@controller3-vm ~]# RabbitMQctl list_queues name slave_pids synchronised
slave_pids|more
```

Listing Queues ...

```
q-agent-notifier-tunnel-update_fanout_bela1840f2d242b9a4a8c80d746c4c5
f [<rabbit@controller3-vm.2.1000.0>, <rabbit@controller2-vm.3.798.0>]
[<rabbit@controller2-vm.3.798.0>, <rabbit@controller3-vm.2.1000.0>]
metering.controller2-vm [<rabbit@controller2-vm.3.1668.0>, <rabbit@controller3-
vm.3.26612.3>] [<rabbit@controller2-vm.3.1668.0>, <rabbit@controller3-
vm.3.26612.3>]
heat-engine-listener.76666c92-e20a-402e-bf66-a5de859e3abb [<rabbit@
controller2-vm.3.2094.0>, <rabbit@controller3-vm.3.26993.3>] [<rabbit@
controller2-vm.3.2094.0>, <rabbit@controller3-vm.3.26993.3>]
```

以三个队列 `q-agent-notifier-tunnel-update_fanout_bela1840f2d242b9a4a8c80d746c4c5f`、`metering.controller2-vm` 和 `heat-engine-listener.76666c92-e20a-402e-bf66-a5de859e3abb` 为例，在三个 RabbitMQ 集群节点中均可以看到这三个队列，并且这三个队列在两个 Slave 节点 `controller2-vm` 和 `controller3-vm` 中都做了镜像。

当 RabbitMQ 服务加入 Pacemaker 并作为其资源对象运行后，Pacemaker 将会对 RabbitMQ 集群进行监控和管理等操作，而这些操作本质上是调用 RabbitMQ 提供的 OCF 脚本来实现的，包括 RabbitMQ 服务的运行状态监控、服务启动和停止、节点 Master 提升和节点 Slave 降级等。图 5-15 描述了 Pacemaker 在提升 RabbitMQ 节点为新的 Master 资源时的操作流程。在 Pacemaker 提升新的 Master 资源时，需要注意的是，每次 Pacemaker 发起的 Promote 操作，都会导致 RabbitMQ 的 OCF 脚本去调用策略文件脚本 `set_RabbitMQ_policy.sh`，并通过该脚本重新设置 RabbitMQ 队列镜像模式或者其他相关的策略。此外，在某个节点提升为 Master 的过程中，其他 RabbitMQ 节点应该保持正常运行，如图 5-15 中的 node-2 节点，在 node-1 提升为 Master 的过程中，node-2 继续运行，并在重启 RabbitMQ 应用后将其加入已成为 Master 的 node-1 节点中。

图 5-16 显示了 Pacemaker 将节点由 Master 降级为 Slave 节点时的操作流程，在 node-1 节点由 Master 降级的过程中，至少要保证一个 RabbitMQ 节点处于运行状态，否则 RabbitMQ 的消息服务将会完全终止，图 5-15 中，node-2 节点继续运行，并在 node-1 节点降级为 Slave 后将其从原来的 Master 节点 node-1 中撤离。

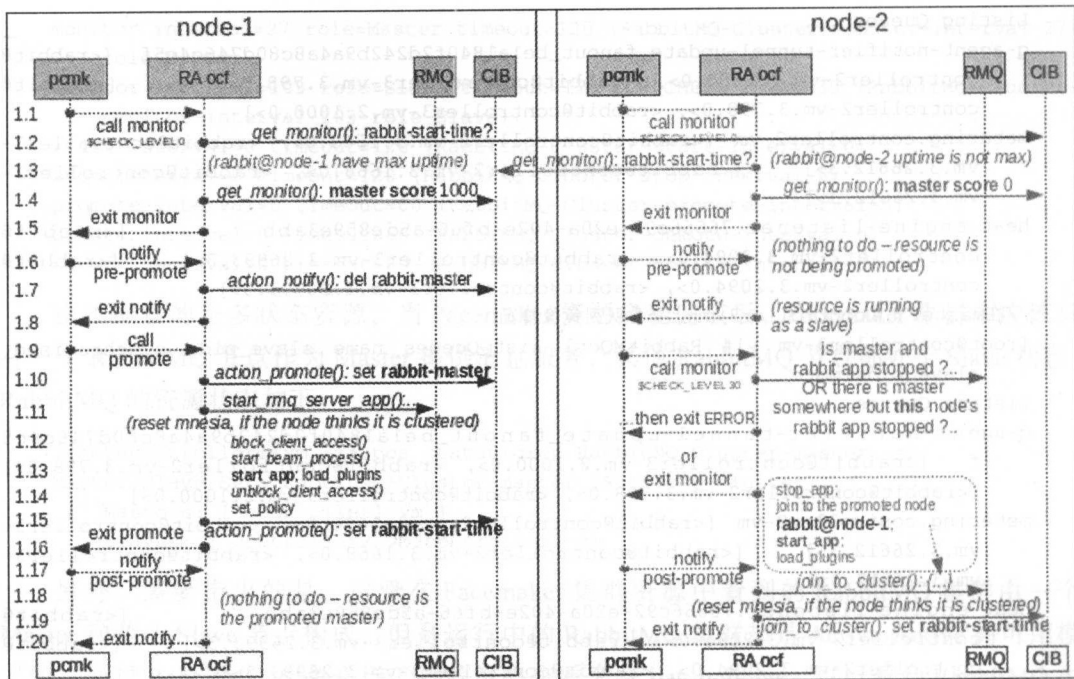


图 5-15 提升一个 RabbitMQ 节点为 Master

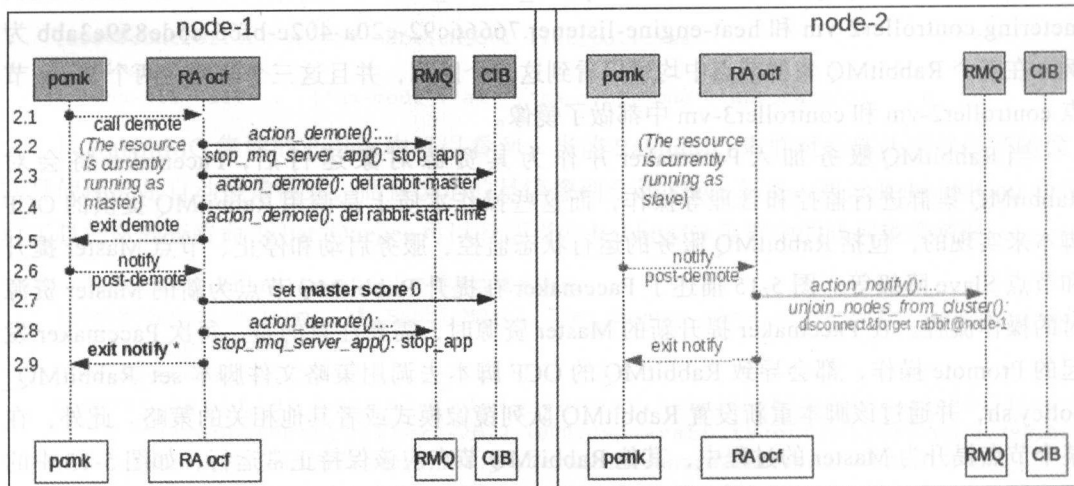


图 5-16 降级一个 RabbitMQ 节点为 Slave

图 5-17 为 Pacemaker 在启动一个 RabbitMQ 节点时的操作流程，图中 node-2 为已经运行的 RabbitMQ 节点，node-1 为正在启动的节点，node-1 启动之后将会立即加入 Master 节点中，而 node-2 节点保持运行不变。在 node-1 节点启动完成后，Pacemaker 可能会丢失部分启动完成后的节点通知事件（post-start notify events），而为了确保这些未收到 post-start 通知事件的节点成功加入 RabbitMQ 集群，Pacemaker 会对正常运行的节点也发起检查通知。

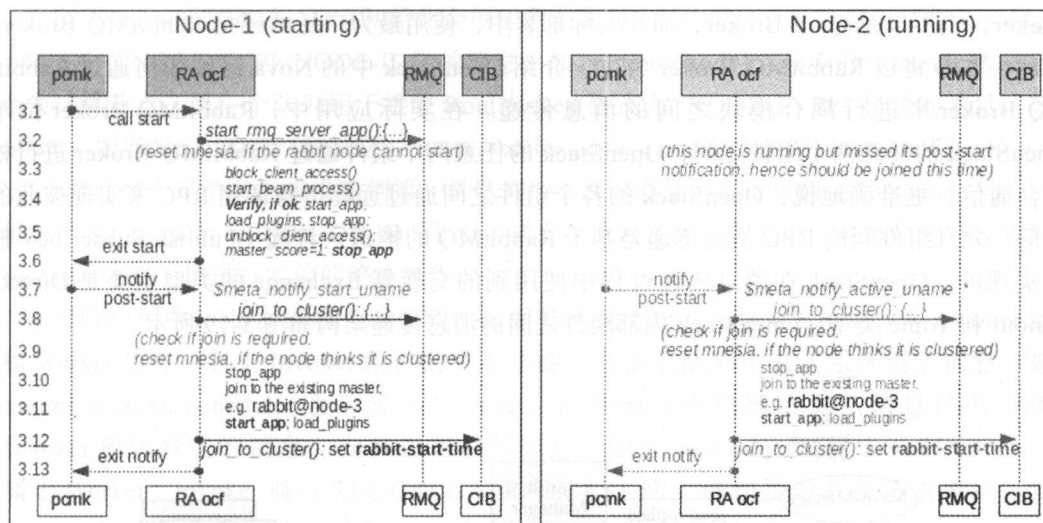


图 5-17 启动一个 RabbitMQ 节点

图 5-18 为 Pacemaker 在停止一个 RabbitMQ 节点时的操作流程，在停止节点时，如果某个运行 RabbitMQ 资源代理的 Corosync 节点没有响应，如 Corosync 进程故障，则 Pacemaker 不会调用停止资源的 post-stop 通知，即 RabbitMQ 节点停止失败。要解决这个问题，需要借助外部的隔离（Fencing）操作，Fencing 操作会强制将故障节点从集群中移除。此外，当节点恢复后，Pacemaker 会将节点自动加入 Corosync 集群和 RabbitMQ 集群中。

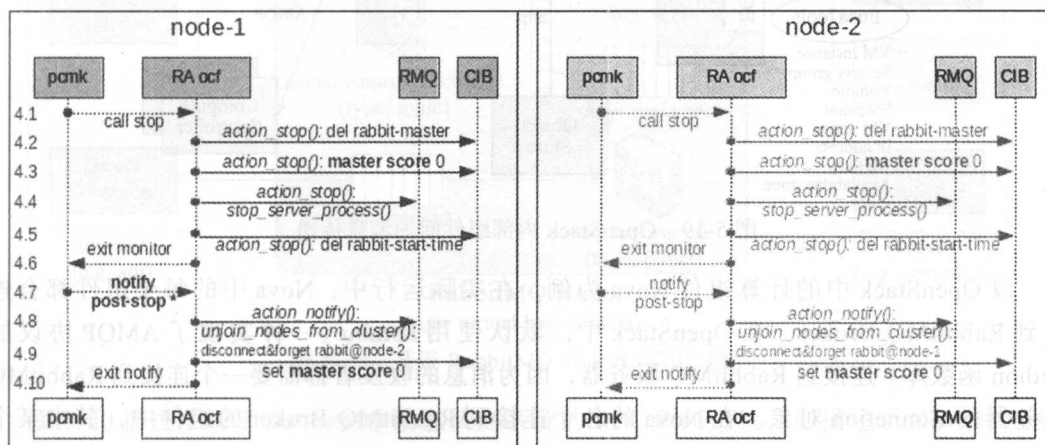


图 5-18 停止一个 RabbitMQ 节点

5.9 RabbitMQ 在 OpenStack 中的应用分析

AMQP 是 OpenStack 云平台采用的一种消息处理机制，AMQP 既可以是 RabbitMQ

Broker，也可以是 Qpid Broker，而在实际部署中，使用最为广泛的还是 RabbitMQ Broker。因此，本节将以 RabbitMQ Broker 为主，介绍 OpenStack 中的 Nova 组件如何通过 RabbitMQ Broker 来进行耦合模块之间的消息传递。在实际应用中，RabbitMQ Broker 位于 OpenStack 的任意两个组件之间，OpenStack 的任意两个组件通过 RabbitMQ Broker 进行松耦合通信。更准确地说，OpenStack 的各个组件之间通过远程过程调用 RPC 来实现彼此的通信，并且组件间的 RPC 消息传递是基于 RabbitMQ 的推送 - 订阅 (Publish/Subscribe) 模式实现的。OpenStack 在消息传递过程中使用到的交换器 Exchange 的类型主要是 Direct、Fanout 和 Topic 类型，OpenStack 内部组件间的消息传递架构如图 5-19 所示。

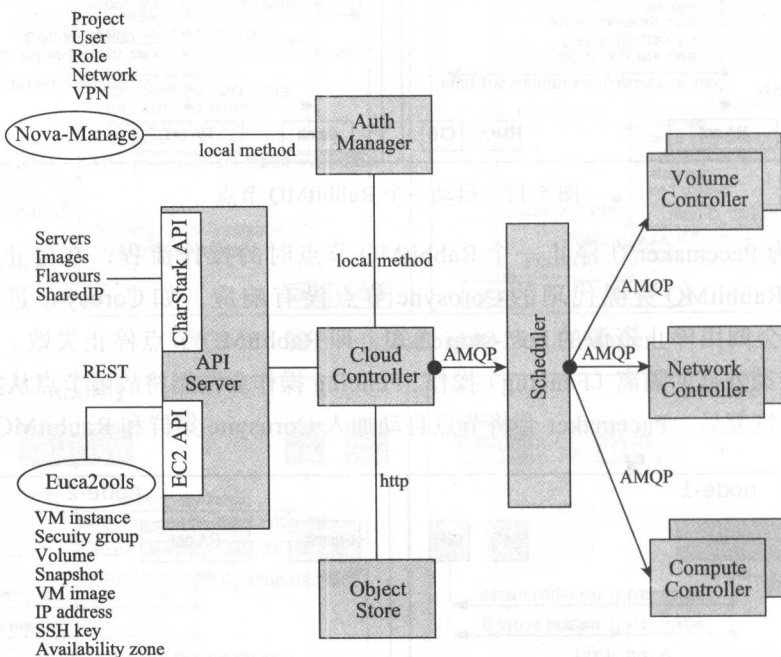


图 5-19 OpenStack 内部组件间的消息传递

以 OpenStack 中的计算组件 Nova 为例，在实际运行中，Nova 中的每个组件都会连接到 RabbitMQ Broker，在 OpenStack 中，默认使用 kombu（一种实现了 AMQP 协议的 Python 函数库）连接到 RabbitMQ 服务器，因为消息的收发者都需要一个连接到 RabbitMQ 服务器的 Connection 对象。在 Nova 的各个连接到 RabbitMQ Broker 的组件中，其中某个组件可能是消息的发送者 Publisher，如 Nova-api 和 Nova-scheduler，也可能是消息的接收者 Consumer，如 Nova-compute 和 Nova-network。此外，某个组件在不同的时刻既可以是 Publisher，也可以是 Consumer。组件发送消息有两种方式，即同步调用 (RPC.CALL) 和异步调用 (RPC.CAST)，当 Nova 发起 RPC.CALL 调用的时候，Nova-api 充当的就是消息的 Consumer，否则是消息的 Publisher。

在启动 Nova 的服务时,启动初始化会创建两个队列,其中的一个队列用于接收 Routing-Key 格式为“NODE-TYPE.NODE-ID”的消息,如 compute.node1,其中的 compute 表示该节点为计算节点;另一个队列用于接收 RoutingKey 格式为“NODE-TYPE”格式的消息,如 compute。如当 Nova 的客户端发送“nova stop instance_name”命令到 Nova-api 时, Nova-api 就会将此命令以消息形式投递到第一种队列中(具体消息的路由投递由 Exchange 操作),从而通过 RoutingKey 中的 NODE-ID(节点主机名)找到目标计算节点,并将命令转发到此 Hypervisor 主机上以进行实例的停止操作。

在实际应用中, Nova 的各个功能模块根据其逻辑功能可以划分为两类,即 Invoker 组件和 Worker 组件。其中 Invoker 组件的主要功能是向消息队列中发送系统请求消息,如 Nova-api 和 Nova-Scheduler 通常属于 Invoker;而 Worker 模块则负责从消息队列中提取 Invoker 模块发送的消息并向其返回应答消息,如 Nova-Compute 和 Nova-Network 通常属于 Worker。Invoker 通过 RPC.CALL 和 RPC.CAST 两个进程发送系统请求消息,然后 Worker 从消息队列中接收消息,并对 RPC.CALL 做出应答响应。Invoker、Worker 与 RabbitMQ 中不同类型的交换器和队列之间的通信关系如图 5-20 所示。在 Nova 的内部组件的 RPC 调用过程中,图 5-20 中的“name:control_exchange”交换器的“control_exchange”应该是 nova, type 为 Topic,此外,该交换器还通过不同的 BindingKey 绑定了多个不同的队列。

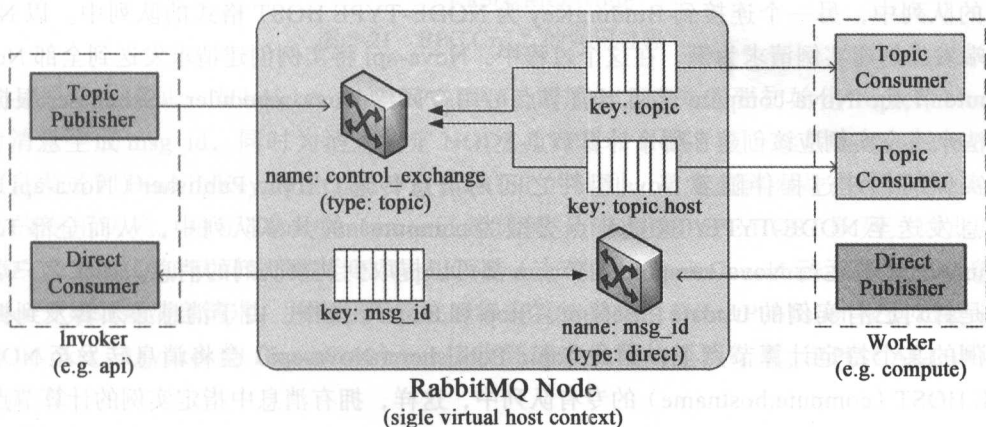


图 5-20 基于 RabbitMQ 的 RPC 调用过程

图 5-20 描述了基于 RabbitMQ 消息队列系统的 RPC 调用过程,即描述了 Nova 中组件之间的 RPC.CALL 调用过程,其中有几个关键功能抽象组件,其解释如下。

- ❑ Topic Publisher: 该对象在 Nova 中的组件发起 RPC.CALL 调用时创建,消息发送出去之后便结束,生命周期短暂,主要用于将消息发送到队列系统中,每个 Topic Publisher 都会连接到 Topic 类型的 Exchange 中。
- ❑ Direct Consumer: 该对象创建于 RPC.CALL 调用之后,专门用来接收 RPC.CALL

调用返回的应答消息，接收到应答消息后便结束。每个 Direct Consumer 连接到一个以 msg_id 为 BindingKey 的专有队列中，同时该队列绑定到一个只接收消息的 RoutingKey 为 msg_id 的交换器 Exchange 中，即该交换器只路由 RoutingKey=BindingKey=msg_id 的消息，而这个消息的 msg_id 被封装在 Topic Publisher 发出的消息中，并且是由一个专门的 UUID 生成器所生成。

❑ Topic Consumer：该对象在 Nova 组件服务启动时创建，在服务结束时候销毁，主要用于接收消息队列中的消息。每个 Worker 都有两个 Topic Consumer，一个连接 BindingKey 为 Topic 的队列，一个连接 BindingKey 为 Topic.host 的队列。每个 Topic Consumer 通过一个独占或者共享的队列与 Topic Publisher 连接到同一个 Topic 类型的 Exchange。

❑ Direct Publisher：该对象创建于 Nova 中的 RPC.CALL 调用返回应答消息时，当 Response 消息发送完成后便结束，与 BindingKey 为特定 msg_id 的 Direct 类型 Exchange 连接。

从生成对象的生命周期来看，图 5-20 中的 TopicPublisher、DirectConsumer 和 DirectPublisher 三个对象是在 RPC.CALL 调用发起时创建的，而 TopicConsumer 是 Nova 的各个服务在启动连接到 RabbitMQ Broker 时创建的，并在服务结束时销毁。Nova 中每一个服务在启动时会创建两个 Topic Consumer 对象，其中一个连接到 BindingKey 为 NODE-TYPE 格式的队列中，另一个连接到 BindingKey 为 NODE-TYPE.HOST 格式的队列中。以 Nova 客户端发出创建实例请求为例，在这个过程中，Nova-api 将实例创建请求发送到全部 Nova-compute 节点，Nova-compute 返回当前节点可用空间给 Nova-scheduler，Scheduler 根据决策算法来决定实例应该创建在哪个计算节点上。

实例的创建过程伴随着 Nova 组件之间的消息传递，Topic Publisher (Nova-api) 会将消息发送至 NODE-TYPE (这里节点类型为 compute) 的共享队列中，从而全部 Topic Consumer (全部运行 Nova-compute 的节点) 都可以提取到共享队列的消息。如果客户端发出的是针对已有实例的 Update、Reboot、Stop 和 Start 等操作，由于消息必须转发到拥有该实例的某个特定计算节点上，因此 Topic Publisher (Nova-api) 会将消息转发至 NOTE-TYPE.HOST (compute.hostname) 的专有队列中，这样，拥有消息中指定实例的计算节点便会接收到此消息。在 Openstack 的 Nova 项目中，主要通过两种 RPC 调用来实现消息传递。

(1) RPC.CALL

RPC.CALL 属于请求 / 响应类型的调用，当请求发送出去以后，需要等待执行结果的响应，这类调用需要指定执行请求的目标对象节点，并等待目标返回的执行结果，RPC.CALL 的调用过程如图 5-21 所示，整个调用过程主要分为以下几个步骤：

1) 初始化一个 TopicPublisher，用以将发送消息到 RabbitMQ 的消息队列。此外，在发送消息之前，需要初始化一个 DirectConsumer 并以 msg_id 作为 Direct 类型 Exchange 的名称，用于等待消息执行后的应答响应。

2) 请求消息被 Exchange 路由到 NODE-TYPE.HOST 消息队列 (图 5-21 中的 topic.host 队列), 然后, 订阅了此队列的相应服务节点 (Nova 中为 compute 节点) 上的 TopicConsumer 会从该队列中获取消息。

3) TopicConsumer 从队列中提取消息后, 服务节点根据消息内容 (调用函数及参数) 调用相应函数完成消息请求处理, 处理完成之后, DirectPublisher 被初始化, 并根据请求消息 msg_id, 将应答消息发送到相应的 Exchange 和消息队列中。

4) 应答消息被位于发起 RPC.CALL 调用方的 DirectConsumer 获取到, RPC.CALL 调用过程完成。

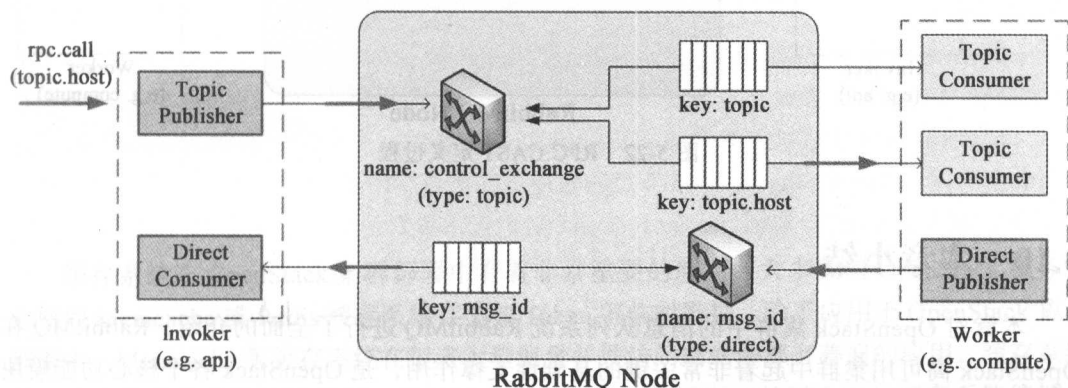


图 5-21 RPC.CALL 的调用过程

由图 5-21 可见, 当 Invoker 发起 RPC.CALL 调用时, 首先需要初始化 TopicPublisher, 并为消息生成 msg_id, 同时为消息指定 NODE-TYPE 和 HOST 值, 然后由 TopicPublisher 将消息发送到 BindingKey 为 NODE-TYPE.HOST 的队列, 通常此队列是由特定 Consumer 定义的专用队列。由于 TopicConsumer 已经订阅了 NODE-TYPE.HOST 队列, 因此进入 NODE-TYPE.HOST 队列的消息会被 TopicConsumer 收到, TopicConsumer 获取请求消息后, Worker 开始处理消息请求, 并在处理完成后将应答结果交由 DirectPublisher 根据 msg_id 返回给 Invoker, Invoker 的 DirectConsumer 接收到请求执行的返回结果, RPC.CALL 调用过程完成。

(2) RPC.CAST

RPC.CAST 属于单向 RPC 请求, 即只将请求发送出去, 无须等待返回执行结果。因此, RPC.CAST 调用不关心请求由哪个服务节点完成, 只需将请求发送到消息队列即可, 而接收消息的队列通常为共享队列, 该队列中的消息可以被某一类型的多个节点 (如 Nova 中的全部计算节点) 接收, RPC.CAST 的调用过程相对简单, 主要由以下两个步骤:

1) Invoker 初始化 TopicPublisher, 并将消息发送到 RabbitMQ 消息服务器中。

2) RabbitMQ 的交换器将消息转发到 NODE-TYPE 类型的共享消息队列 (图 5-22 中的 topic 队列) 中, 并被相应服务节点 (Nova 中为 Compute 节点) 的 TopicConsumer 获取,

TopicConsumer 提取到订阅队列的消息后, Woker 便开始处理消息请求, 至此, RPC.CAST 的过程已经完成, Invoker 不会再等待 Woker 返回请求消息执行后的结果。

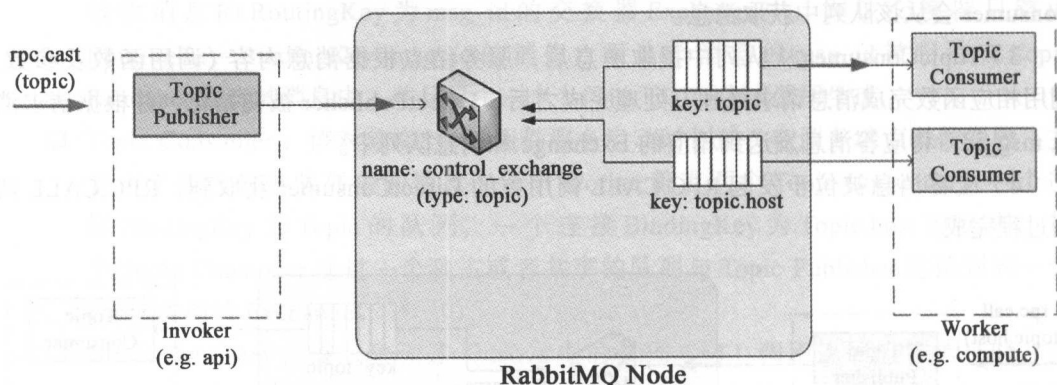


图 5-22 RPC.CAST 定义过程

5.10 本章小结

本章对 OpenStack 集群中的消息队列系统 RabbitMQ 进行了全面的介绍, RabbitMQ 在 OpenStack 高可用集群中起着非常关键的基础性支撑作用, 是 OpenStack 各个核心功能模块彼此交互通信的关键。本章对 AMQP 和 RabbitMQ 的专业术语进行了详细介绍, 同时介绍了 RabbitMQ 的简单配置和集群配置, 并从高可用性的角度, 分析演示了 RabbitMQ 的高可用集群配置。此外, 本章还以 Nova 组件为例, 讲解了 OpenStack 使用 RabbitMQ 进行消息传递的过程。通过本章的学习, 读者应该能够对 OpenStack 中的消息队列系统具有全局和理论上的认识, 同时也应该能够配置高可用的消息队列集群, 并能够对常见的消息通信问题进行故障排查。

集群缓存系统

缓存系统在 OpenStack 集群部署中有着非常重要的应用，大多数的 OpenStack 服务都会使用 Memcache 或 Redis 缓存系统存储如 Token 等临时数据。除了应用于 OpenStack 集群部署中，Memcache 等缓存系统在很多大型海量并发访问网站中都有普遍的应用，缓存系统可以认为是基于内存的数据库，相对于后端大型生产数据库，基于内存的缓存系统能够提供快速的数据访问操作，从而提高客户端的数据请求访问，并降低后端数据库的访问压力。对于关系型数据库，尤其是大型关系型数据库，如果对其进行每秒上万次的并发访问，并且每次访问都在一个有上亿条记录的数据表中查寻某条记录，其效率将会非常低，对数据库而言这也是无法承受的过程。

缓存系统的使用，可以很好地解决大型并发数据访问所带来的效率低下和数据库压力等问题，缓存系统将经常使用的活跃数据存储在内存在中，避免了访问重复数据时数据库查询所带来的频繁磁盘 IO 和大型关系表查询时的时间开销，因此，缓存系统几乎是大型网站的必备功能模块。本章将会介绍时下应用最为广泛的 Memcache 和 Redis 缓存系统，并结合其在 OpenStack 高可用集群中的部署事例，讲解其在高可用集群中的工作原理和缓存数据高可用的配置方法，通过本章的学习，读者应该能够了解主流缓存系统软件的工作原理和使用配置方法。

6.1 Memcache 缓存系统

6.1.1 Memcache 缓存概述

Memcache 是互联网领域使用极为广泛的开源分布式对象缓存系统，通过 BSD License

授权发行，最初由互联网公司 DangaInterActive 所开发，其目的是为了创建内存缓存系统来应对其网站 LiveJournal.com 所承载的巨大访问流量。由于 LiveJournal 网站每天超过 2000 万的页面访问量给其数据库施加了巨大的压力，因此 DangaInterActive 的 Brad Fitzpatrick 便着手设计了 Memcache 缓存系统。作为一个极为实用的开源系统软件，Memcache 在降低 LiveJournal.com 网站数据库负载的同时，也被众多高并发访问网站采用为主流的缓存解决方案。

Memcache 利用系统内存对客户端经常进行反复读写和访问的数据进行缓存，来减轻后端数据库的访问负载和提高客户端的数据访问效率。Memcache 中缓存的数据经过 HASH 之后被存放到位于内存上的 HASH 表内，而 HASH 表中的数据以 Key-Value 的形式存放。由于 Memcache 没有实现访问认证及安全管理控制，因此在面向 Internet 的系统架构中，Memcache 服务器通常位于用户的安全区域。多个 Memcache 节点服务器可以组成 Memcache 集群，在 Memcache 集群中，Memcache 客户端的 API 通过 32Bit 的循环冗余校验码（CRC-32）将存储数据的键值对进行 HASH 计算后，存储到不同的 Memcached 节点服务器上，而当 Memcached 服务器节点的物理内存剩余空间不足时，Memcached 将使用最近最少使用算法（LRU, LeastRecentlyUsed）对最近不活跃的数据进行清理，从而整理出新的内存空间存放需要存储的新数据。

Memcache 最初的设计理念就是通过简单的架构设计和程序编码来实现强大的集群数据缓存功能，因此，Memcache 在安装配置、集群部署以及客户端的使用配置上都非常简便。同时，Memcache 在解决大规模集群数据缓存的诸多难题上又具有非常明显的优势，并且还易于进行二次开发，因此越来越多的用户将其作为集群缓存系统。此外，Memcache 开放式的 API 使得大多数的程序语言都能使用 Memcache，如 Java、C/C++/C#、Perl、Python、PHP、Ruby 等各种流行的编程语言。最后需要指出的是，很多刚接触 Memcache 的技术人员可能会混淆 Memcache 和 Memcached 的概念，其实二者一个是项目名称，另一个是运行在系统中的进程名称，本质上是同一个事物。在这里，Memcache 是开源项目名称，Memcached 是运行在服务器端的 Memcache 进程（Memcache Deamon）。

由于 Memcache 的诸多优势，其已成为众多开源项目的首选集群缓存系统。在 OpenStack 的集群部署中，大部分子项目都会用到内存缓存系统进行客户端访问数据的缓存，从而提高访问速率并减轻数据库的负载压力，而 OpenStack 官方社区均指定了 Memcache 作为其众多子项目部署的内存缓存系统。如 OpenStack 的 Keystone 身份验证项目就会利用 Memcache 来缓存租户的 Token 等身份信息，从而在用户登录验证时无需查询存储在 MySQL 后端数据库中的用户信息，这在数据库高负荷运行下的大型 OpenStack 集群中能够极大提高用户的身份验证过程。再如 Web 管理界面 Horizon 和对象存储 Swift 项目也都会用到 Memcache 来缓存数据以提高客户端的访问请求响应速率。

6.1.2 Memcache 的工作原理

Memcache 缓存系统在工作流程的设计上比较简单，其主要思想就是 Memcache 的客户

端根据用户访问请求中的 Key，到 Memcached 服务器的内存 HASH 表中获取对应此 Key 的 Value，Memcache 客户端取到值之后直接返回给用户，而不用再到数据库中进行数据查询。当然，内存 HASH 表中不可能预知和存储全部客户端需要的 Key-Value 值对，因此，如果在 Memcache 服务器中没有找到与请求中的 Key 对应的 Value，则转向后端数据库进行查询，并将查询到的 Value 与 Key 进行 HASH 后存入 Memcached 服务器中，从而保证曾经存储过的数据一定能被查询到，并将数据库中查询到的数据缓存到 Memcache 服务器中，从而提高下一次缓存查询的命中率。Memcache 缓存系统的工作流程如图 6-1 所示。

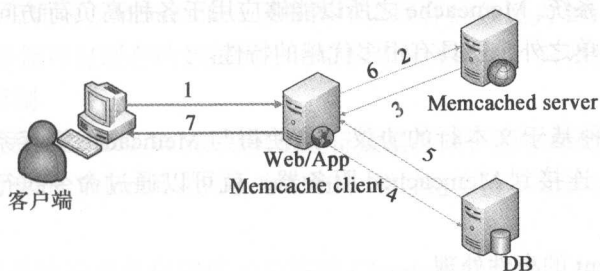


图 6-1 Memcache 的工作流程

图 6-1 描述了 Memcache 缓存系统的大致工作流程，Memcache 的整个工作流程被分为七步实现，但是并非每次查询都需要经历七个步骤，这需要根据是否在缓存中命中请求 Key 对应的 Value 来决定。Memcache 缓存系统的工作原理可以描述如下。

- 用户终端向 Web 服务器发起请求（Web 服务器中部署有 Memcache 客户端），Web 服务器中的 Memcache 客户端向 Memcached 服务器发起查询请求，查询目标为用户请求中包含的 Key 所对应的 Value，请求响应流程如图 6-1 中的步骤 1 和 2 所示。
- 如果 Memcache 客户端在 Memcached 服务器中查询到与请求 Key 对应的 Value，则直接返回结果，本次数据查询过程结束，即整个过程不会访问数据库。通常，如果客户端针对特定 Key 发起的查询已不是首次查询，采取的便是这类请求响应过程，因为 Memcached 服务器缓存中总会命中曾经查询过的 Key-Value 对。请求响应流程如图 6-1 中的步骤 1、2、3 和 7 所示。
- 如果经过图 6-1 中的步骤 1 和 2 没有查询到数据，则表明 Memcached 服务器没有缓存请求中的 Key 对应的 Value，即本次缓存查询未命中，但是查询到此并不结束，而是转向数据库中继续查询该值，并将查询到的数据返回给客户端，同时将该数据缓存到 Memcached 服务器的 HASH 表中，以便客户端再次发起相同请求时，该值在 Memcached 服务器中能够直接命中，而无需再次查询数据库。通常，如果客户端针对特定的 Key，是首次发起查询，则采用的便是这类请求响应过程。请求响应流程如图 6-1 中的步骤 1、2、4、5、7 和 6 所示。
- 当数据库中存放的永久数据发生更新时，Memcached 服务器中缓存的值也需要同时更新，从而保证 Memcached 缓存服务器中的数据与数据库中的真实值具有一致性。

□ 如果 Memcached 服务器中分配给 Memcached 使用的内存空间耗尽, 则 Memcache 缓存系统将使用 LRU 算法和数据的到期失效策略清理非活跃的冷数据, 在这个过程中, 已达到策略设置中的保存期限的数据将会首先被清除, 然后再清除最近最少使用的数据。

6.1.3 Memcache 的功能特点

Memcache 缓存系统由于其简单实用的设计, 使得其在大规模高并发访问的集群系统中被广泛用作内存缓存系统, Memcache 之所以能够应用于各种高负荷访问流量的集群场景中, 除了其使用和部署简单之外, 还具有很多优越的特性。

(1) 协议简单

Memcache 是一种基于文本行的协议, 这使得与 Memcache 缓存系统的交互变得很简单, 只需通过 Telnet 连接到 Memcached 服务器, 就可以通过命令执行数据的存取和访问操作。

(2) 基于 Libevent 的事件处理

Libevent 是通过 C 语言开发的程序库, Libevent 将 BSD 系统的 Kqueue 和 Linux 系统的 Epoll 等事件处理功能封装成一个统一的接口, 与传统的数据库 Select 查询语句相比, 使用 Libevent 极大提高了性能。

(3) 内存数据存储管理

Memcache 缓存系统的所有数据均保存在内存中, 数据存取速率比硬盘 IO 要快很多。当分配给 Memcached 的内存耗尽后, Memcache 缓存系统自动根据数据失效日期进行失效数据的清理, 同时利用 LRU 算法对最近最少使用数据进行清理。但是, 由于数据全部储存在内存中, 如果重启 Memcached 服务器, 则储存数据会丢失。

(4) 节点相互独立的分布式集群

Memcache 缓存系统可以由多个 Memcached 服务器节点构成, 但是这些 Memcached 服务器节点彼此之间不进行心跳通信, 也就是说任意两个节点之间不会进行数据同步, 同时一个 Memcached 节点故障后其上存储的数据也不会转移到其他节点构成高可用。Memcached 服务器节点彼此之间独立存取数据, 增加 Memcached 服务器数量的目的只是以 Scale-out 的形式扩容整个 Memcache 缓存系统的容量, 增加节点数量并不能为整个集群带来数据的高可用性。

由于 Memcache 缓存系统具有上述优劣并存的功能特点, 因此, 在使用 Memcache 作为缓存系统时, 一定要考虑清楚使用 Memcache 可能会带来哪些不利因素, 这样才能设计适当的解决方案来应对这些可能的故障情况。通常, Memcache 缓存系统在使用过程中可能会存在下述需要考虑的问题。

(1) Memcached 服务器单点故障

由于 Memcache 缓存系统中的每个 Memcached 服务器节点独立存取数据, 彼此之间不

存在数据的镜像同步机制，因此，如果某一个 Memcached 服务节点故障或者重启，则该 Memcached 节点上缓存在内存中的全部数据均会丢失，丢失缓存数据后，访问将无法在缓存中命中任何 Key，所有 Key 的访问都需要到数据库重新查询一遍，同时将查询的数据再缓存到 Memcached 服务器的内存缓存中，即 Memcached 服务器每重启一次，其储存的数据就要被重新缓存一次。

（2）存储空间限制

Memcache 缓存系统的数据存储在内存中，内存数据的存储必然受到寻址空间大小的限制，对于 32 位的系统而言，Memcached 服务器可以缓存的数据为 2GB，对于 64 位系统而言，Memcached 服务器可以缓存的数据大小可以认为是无限的，只要物理存储足够大即可。

（3）存储单元限制

Memcache 缓存系统以 Key-Value 为单元进行数据存储，能够存储的数据 Key 尺寸大小为 250 字节，能够存储的 Value 尺寸大小为 1MB，超过这个值不允许存储。

（4）数据碎片

Memcache 缓存系统的内存存储单元是按照 Chunk 来分配的，这意味着不可能所有存储的 Value 数据大小都正好等于一个 Chunk 的大小，因此必然会造成内存数据碎片而浪费存储空间。

（5）利旧算法局限性

Memcached 缓存系统的 LRU 算法并不是针对全局空间的存储数据的，而是针对 Slab 的，Slab 是 Memcache 中具有同样大小的多个 Chunk 集合。

（6）数据访问安全性

Memcache 缓存系统的 Memcached 服务器端并没有相应的安全认证机制，通过非加密的 Telnet 连接即可对 Memcached 服务器端的数据进行各种操作。

6.1.4 Memcache 集群概述

Memcache 集群的配置极为简单，因为 Memcache 集群无需在 Memcached 服务器端进行配置，数据的分布存储完全取决于 Memcache 客户端的节点选取算法，即数据在集群节点中的存储与提取完全取决于客户端的节点选取算法，而不是服务器端集群配置的。在 Memcache 集群中，服务器端只需运行 Memcached 服务，客户端只需在应用程序的配置文件中指定要访问的 Memcached 服务器节点 IP 地址和端口组成的列表。Memcache 尽管是分布式缓存服务器系统，但服务器端的集群节点之间并没有太多的交互，各个服务器端节点之间独立存取数据，即 Memcached 服务器节点之间不会互相进行心跳检测，或者进行彼此数据的拷贝镜像。在 Memcache 集群缓存系统中，数据的存储过程如图 6-2 所示，数据的提取过程如图 6-3 所示。

当应用程序向 Memcache 集群存储数据时，Memcache 客户端会根据请求存储数据的 Key-Value 值对，利用一定算法从 Memcached 集群存储节点中选取某一个 Memcached

服务器节点来存储数据，当算法选定该节点后，对应 Key 的 Value 就会被存储到该节点中。图 6-2 中 Key 为 “warrior”，Value 为 “data”，算法在节点列表选取到的存储节点为 Memcached1 节点，因此，Key 为 “warrior” 的 Value 值 “data” 将被存储到 Memcached1 中。

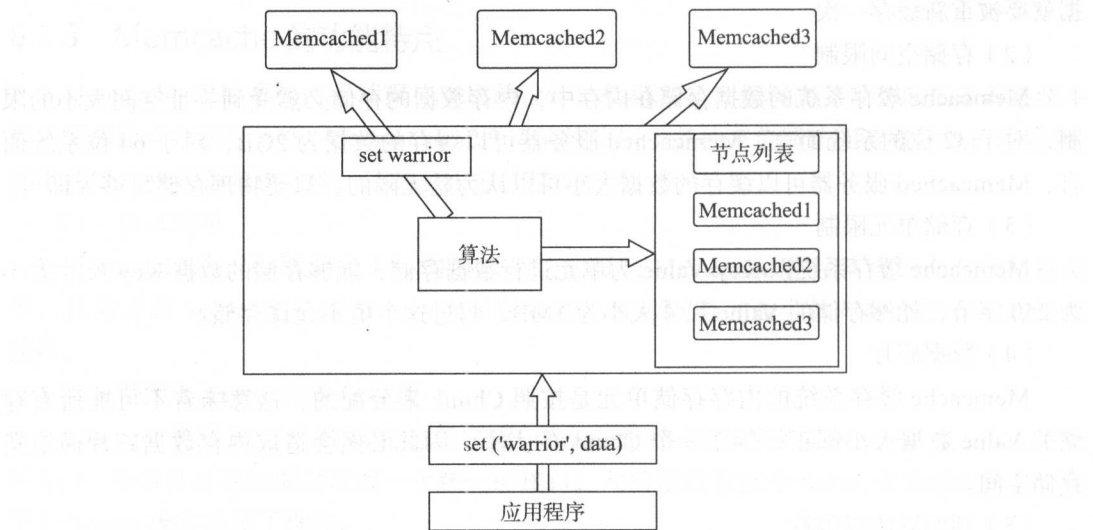


图 6-2 Memcache 集群缓存系统数据存储过程

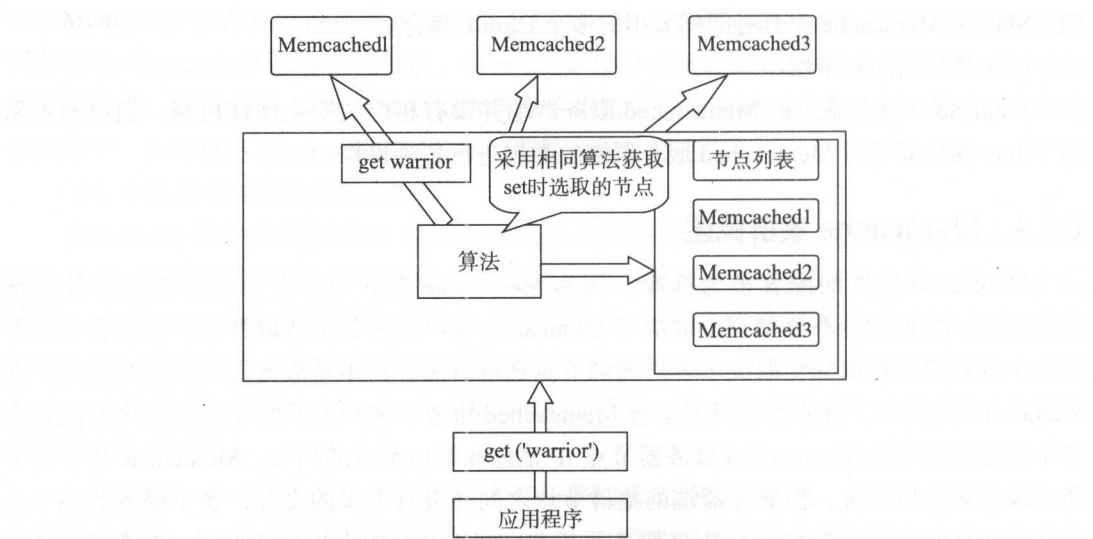


图 6-3 Memcache 集群缓存系统数据提取过程

当应用程序向 Memcache 集群提取数据时，Memcache 客户端会根据请求提取数据的 Key 值，采用与存储该 Key-Value 值时完全一样的算法来选取存储该 Key 值的 Memcached 服务器节点，理论上，如果 Key 值不变，则选取到的节点就是存储该 Key 值

的节点。当找到存储 Key 值的 Memcached 节点后, Memcache 便会到该节点提取对应的 Value 并返回给客户端。图 6-3 中, Key 仍然为 “warrior”, 由于使用存储此 Key 时使用的算法相同, 从节点列表中选取得到的 Memcached 节点一定也是 Memcached1, 因此, 到 Memcached1 节点中提取对应的 Value 值 “data”。

从图 6-2 和图 6-3 中 Memcache 集群的数据存取过程可以看出, Memcache 客户端在选取缓存节点时所采用的算法对数据的最终存储位置有着决定性的作用, 而节点选取算法也是 Memcache 缓存系统中最重要的部分。在通常使用时, Memcache 缓存系统中最常使用的两个算法分别是余数 HASH 算法和一致性 HASH 算法。

1. 余数 HASH 算法

在 Memcache 缓存系统中, 余数 HASH 算法就是将需要存储的 Key-Value 值对的 Key 字符通过 HASH 算法后得到 HASH Code, 这里的 HASH Code 是一个数字, 而 HASH 算法的作用就是将一个字符串通过 HASH 后得到一个数值。通常, 相同的字符串经过相同的 HASH 算法后一定会得到相同的 HASH Code, 因此, 存储和提取相同 Key 的 Memcached 目标节点一定是同一个节点。字符串 HASH 具有以下几个特性:

- 不同字符串本身不会生成相同的 HASH Code, 但是如果将 HASH Code 再次取余作为最终 HASH Code, 则不同字符串的 HASH Code 可能会相同, 即所谓的 HASH 冲突。
- 相同字符串不可能具有不同的 HASH Code。
- 相同字符串每次生成的 HASH Code 一定相同。

在图 6-2 中, Key 字符串为 “warrior”, 假设该 Key 经过 HASH 之后得到的 HASH Code 为 100, Memcache 集群缓存系统中有 3 个 Memcached 节点, 那么余数 HASH 就是将 HASH Code 100 除以节点数目 3, 然后取余得到的结果, 这里结果为 1, 这样选取到的 Memcached 服务器节点为编号为 1, 即 Memcached1 节点。因为不同的 Key 值得到的 HASH Code 是不同的, 这样取余后得到的结果是比较随机均匀的, 数据就会随机存储到不同的 Memcached 服务器节点上, 因此, 如果在一个系统架构中, Memcached 服务器节点数目比较固定, 使用过程中无需扩容节点, 则余数 HASH 算法可以很好地满足 Memcache 集群中的节点选取过程。

然而, 如果用户的业务系统访问量不断增加, 当前的 Memcache 缓存系统已经不够使用而造成了数据库的访问压力增大, 客户端请求响应变得迟钝, 这时用户通常需要扩容 Memcached 服务器节点的数目, 而当用户增加了 Memcached 节点数目后, 余数 HASH 便会出现不能命中缓存的问题。例如, 当增加 3 个 Memcached 节点后, 现在 Memcached 服务器节点数目为 6, 而字符串 “warrior” 的 HASH Code 仍然为 100, 可余数 HASH 的结果却为 2, 而不是之前的 1, 即 Memcache 客户端在提取 “warrior” 这个 Key 的值时, 算法将会路由到 Memcached2 节点去, 但是扩容之前该 Key 的 Value 是存储在 Memcached1 节点中的, 因此客户端将不能在缓存系统命中此 Key。

为了进一步说明节点变化对 Memcache 缓存系统命中率造成的影响，现假设有 20 个 Key，其对应的 HASH Code 分别为 0-19（如表 6-1 所示），原有 Memcached 节点数目为 3，增加一个节点后变为 4 个 Memcached 节点，则对应相同的 HASH Code（即相同的 Key），缓存系统节点增加后选取的 Memcached 服务器节点如表 6-2 所示。

表 6-1 3 个 Memcached 节点时 HashCode 与节点的对应关系

Hashcode	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
选取的服务器	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1

表 6-2 4 个 Memcached 节点时 HashCode 与节点的对应关系

Hashcode	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
选取的服务器	<u>0</u>	<u>1</u>	<u>2</u>	3	<u>0</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>0</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>0</u>	<u>1</u>	<u>2</u>	<u>3</u>	0	1	2	3

从表 6-2 可以看出，增加节点后，只有 HASH Code 为 0、1、2 和 12、13、14 的 Key 能够被准确地命中（图中加粗和下划线标注部分），即命中率降低了 70%，因此节点增加后，一半以上的数据都无法命中。目前，解决余数 HASH 算法中节点的扩容所带来的命中率极大降低的问题，并没有十分有效的方案，通常还是以高成本的人工投入为主。例如，运维人员申请维护窗口，在夜晚进行 Memcached 节点的扩容，然后重启 Memcache 集群系统，使集群中缓存的数据全部失效，然后业务人员模拟访问请求来预热缓存系统，以将通常使用频率较高的数据重新缓存到 Memcache 集群缓存系统中，即让缓存系统中的数据重新分布以恢复命中率。

2. 一致性 HASH 算法

一致性 HASH 通过将 Memcached 节点和 Key 字符串 HASH 后放到一个 HASH 环上，然后 Key 字符串的 HASH 值再以顺时针的方式找到距其最近的节点 HASH 值，该节点 HASH 值对应的节点便是 Key 的 Value 存储节点。如图 6-4 所示，实线所绘制的大圆圈为 HASH 环，实心大圆圈代表三个 Memcached 节点 HASH 后在 HASH 环上的位置，实心小圆圈代表 Key 字符串 HASH 后在 HASH 环上的位置，图中 HASH Code 为 1 的 Key 将存储到 node2 上，HASH Code 值为 2 和 3 的 Key 将存储到 node3 上，HASH Code 为 4 和 5 的 Key 将存储到 node1 上。

如果 Memcache 集群节点数目变为 4，则一致性 HASH 环将会发生变化，图 6-5 显示了在图 6-4 中增加一个节点后，4 节点的一致性 HASH 环。

从图 6-5 中可以看到，当 Memcached 节点由 3 变为 4 后，只有 HASH Code 为 4 的 Key 受到了影响，该 Key 本来应该存储在 node1 节点上，因为增加了 node4 节点，而被重新缓存到了 node4 节点中。从整体上看，一致性 HASH 确实也会影响到 Key 值的命中率（将近 20%），但是相比余数 HASH 对命中率的影响动辄就是 50% 以上，一致性 HASH 对缓存命中率的影响要小很多。并且随着缓存系统中节点数目的增加，这种影响将不断减小，

或者说随着缓存系统节点数目不断增加,缓存命中率将不断提高,在这一点上,缓存命中率与节点数目的增加实现了一致性。

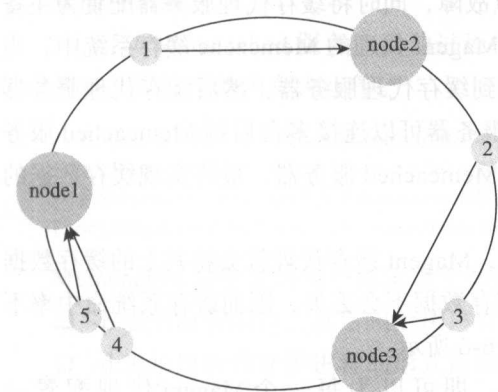


图 6-4 3 个 Memcached 节点时的一致性 HASH 环

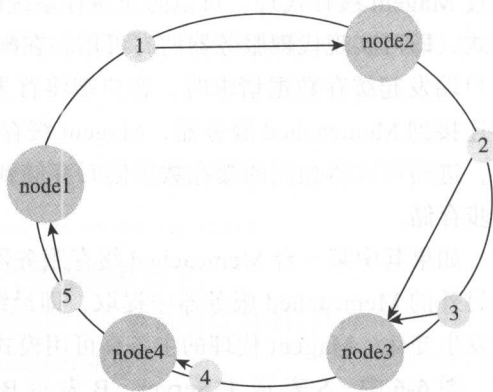


图 6-5 4 个 Memcached 节点时的一致性 HASH 环

对于 Memcache 缓存系统而言,究竟选择余数 HASH 还是一致性 HASH,需要根据 Memcache 缓存系统的实际使用情况而定。如果在系统架构初期就有足够预算配置充足的缓存节点,则可选用余数 HASH 算法,例如公司内部自用的私有云或者 Web 访问站点。而对于大型互联网 Web 系统,由于整个架构弹性很大,根据业务需求可能需要不断增加缓存节点以降低日夜增大的访问量对数据库的压力,因此面向商业系统的缓存架构最好选择一致性 HASH 算法。

6.1.5 Memcache 集群高可用

由于 Memcache 集群中的 Memcached 服务器节点之间没有任何心跳检测机制,并且也不进行任何数据镜像备份,所以当任何一个 Memcached 服务器节点出现故障时,都会降低整个 Memcache 集群缓存系统的命中率。尽管节点故障后可以正常恢复,但是缓存在其中的数据已经丢失,并且节点恢复重新加入集群时,由于集群节点数目再次发生变化,所以不管采用的是余数 HASH 还是一致性 HASH 算法,都会再次降低缓存命中率,即集群节点的退出和重新加入可能会反复降低集群的缓存命中率。

当然,只要最终存储数据的数据库服务器正常运行,缓存节点中丢失的数据还可以重新缓存,因此,正常情况下,丢失 Memcache 集群中某个缓存节点的数据,除了影响缓存命中率之外,并不会影响缓存系统的正常使用,同时丢失的缓存数据也可以从数据库中提取并重新缓存。但是,如果用户仍然无法忍受缓存系统的这种单点故障,尤其在大规模访问的业务高峰期,Memcached 节点的故障可能会使得某些请求反应相当缓慢,并极大影响客户的使用体验,为了解决 Memcache 集群系统的单点故障,用户需要借助其他软件来实现 Memcache 集群系统的高可用。

Memcache 缓存系统最常使用的 HA 方式是 Memcached 节点代理,其中最常使用的代

理软件为 Magent。Magent 是一款开源并且简单实用的 Memcached 服务器代理软件，其开源项目网址为 <http://code.google.com/p/meMagent> 和 <https://github.com/wangmh/meMagent>。通过 Magent 缓存代理，可以防止缓存系统的单点故障，同时将缓存代理服务器配置为主备模式，即可实现代理服务器的高可用。在配置有 Magent 代理的 Memcache 缓存系统中，当客户端发起缓存数据请求时，客户端将首先连接到缓存代理服务器，然后缓存代理服务器再连接到 Memcached 服务器，Magent 缓存代理服务器可以连接多台后端 Memcached 服务器，进而可以将相同的缓存数据同时存储到多个 Memcached 服务器，最终实现缓存数据的同步存储。

如果其中某一台 Memcached 缓存服务器宕机，Magent 缓存代理就会将其上的缓存数据从另外的 Memcached 服务器中提取，即最终的缓存数据不会丢失，因而缓存系统命中率不会发生变化。Magent 代理的主备高可用模式如图 6-6 所示。

图 6-6 中，S 表示主 Server，B 表示 Backup，即可以为每一个 Magent 代理配置一个后端 Memcached 节点作为缓存数据的主服务器，同时还可以为其配置一个或多个后端 Memcached 节点作为缓存数据的备用服务器，而同一个 Memcached 后端服务器既可以是某个 Magent 的主 Server，同时也可以是其其他 Magent 的 Backup。图 6-6 的高可用架构中配置了两个 Magent 代理节点，两个 Memcached 后端服务器节点，每个 Magent 节点又分别代理两个 Memcached 节点，其中的两个 Memcached 节点以主备形式缓存数据，而应用系统客户端通过访问 Magent Pool 来间接调用 Memcache 缓存系统。

图 6-6 的架构实现起来非常简单，两个 Memcached 节点的硬件及系统配置只需两台安装有 Linux 系统的 X86 服务器，每台服务器上分别安装有 Magent 和 Memcached 软件，同时将两个服务均设为开机自启动，稍做简单配置即可实现 Magent 代理的高可用缓存系统。在图 6-6 中，任何一台服务器故障后，由于两台服务器同时缓存有相同数据，因此 Magent Pool 能够从另一台服务器正常获取到 Memcached 缓存的数据，从而实现 Memcache 缓存系统的高可用性。当然，如果两台机器都同时宕机，那么缓存系统中的数据仍然会丢失。如果要保证更高的可用性，用户可以使用三台、四台或者更多的服务器来构建更强壮的 Magent 代理高可用 Memcache 缓存系统。

Magent 代理软件的配置非常简单，在系统中安装完成 Memcached 软件和 Magent 代理软件之后，在启动 Memcached 服务时指定服务的监听地址及端口，并在启动 Magent 服务时指定后端 Memcached 服务器的主备关系。以实现图 6-6 的高可用架构为例，假设两个 Linux 系统已经成功安装 Memcached 和 Magent 软件，两个系统的主机名分别为 controller1-vm 和 controller2-vm，对应的主

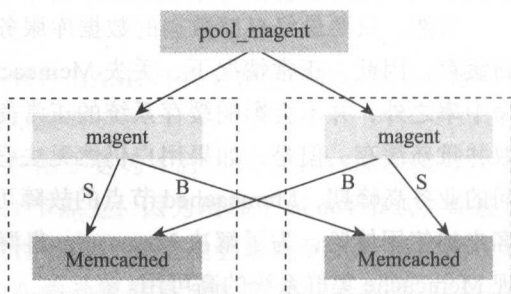


图 6-6 Magent 缓存代理主备高可用架构

机 IP 分别为 192.168.142.110 和 192.168.142.111。并设定 Memcached 的监听端口为默认的 11211 端口, Magent 的监听端口也为 11211, 则 Magent 代理的高可用缓存系统的配置用两个步骤即可实现。

1) 启动 Memcached 服务, 启动过程中用户可以指定多个自定义参数, 如监听的 IP 地址及端口, 可使用的内存大小等。

```
//启动controller1-vm节点的Memcached服务
[root@controller1-vm ~]#./usr/bin/Memcached -d -m 64 -p 11211
//启动controller2-vm节点的Memcached服务
[root@controller2-vm ~]#./usr/bin/Memcached -d -m 64 -p 11211
```

Memcached 命令允许用户指定多个参数, 各个参数解释如下:

- -p 表示监听端口, 其后跟上用户自定义的端口号, 默认端口为 11211;
- -d 表示以后台守护进程的形式启动 Memcached 服务;
- -u 表示以 root 用户启动 Memcached 服务, 默认不能用 root 用户启动;
- -P 表示进程的 pid 文件的存放位置;
- -l 表示监听 IP 地址, 其后跟上用户自定义的监听 IP 地址, 默认监听所有 IP;
- -m 表示内存, 后跟分配内存大小, 以 MB 为单位, 默认为 64M;
- -c 表示最大并发连接数, 默认为 1024;
- -f 表示块大小增长因子, 默认是 1.25;
- -M 表示内存耗尽时返回错误给客户端, 而不用 LRU 算法清理数据。

2) 启动 Magent 服务, 在启动 Magent 之前, 必须保证 Memcached 服务已经正常启动运行, 在启动 Magent 服务的同时, 直接为 Magent 代理指定主备 Memcached 后端缓存服务器。

```
//启动controller1-vm节点上的Magent服务, 并将本机设置为Magent代理的Memcached主节点, 将
controller2-vm设置为backup节点
[root@controller1-vm ~]#./usr/bin/Magent -n 4096 -l 192.168.142.110 -p 11200 -s
192.168.142.110:11211 -b 192.168.142.111:11211
//启动controller2-vm节点上的Magent服务, 并将本机设置为Magent代理的Memcached主节点, 将
controller1-vm设置为backup节点
[root@controller2-vm ~]#./usr/bin/Magent -n 4096 -l 192.168.142.111 -p 11200 -s
192.168.142.111:11211 -b 192.168.142.110:11211
```

Magent 命令也允许用户指定多个启动参数, 各个参数解释如下:

- -u 表示启动服务的用户 ID;
- -g 表示启动服务用户的组 ID;
- -p 表示启动后的监听端口, 默认 11211, 0 表示不监听 TCP 协议;
- -s 表示当前 Magent 代理的后端主 Memcached 服务的地址和端口;
- -b 表示当前 Magent 代理的后端备份 Memcached 服务的地址和端口;
- -l 表示监听的 IP 地址, 默认监听全部 IP 地址;

- ❑ `-n` 表示允许的最大并发数，默认是 4096；
- ❑ `-D` 表示服务以非后台运行方式启动；
- ❑ `-k` 表示使用 ketama Key 分配算法；
- ❑ `-f` 表示进程文件存放路径；
- ❑ `-i` 表示单个 Memcached 服务器运行时允许的活动连接数，默认是 20。

当 Memcached 和 Magent 代理服务都配置启动完成后，应用程序客户端便可通过 Magent 代理到 Memcache 缓存系统中存取数据。在 Magent 代理中，尽管其中一个节点故障后不影响缓存数据的访问命中率，但是故障节点重启后，其上的缓存数据仍然会丢失（因为缓存数据全存储在内存中，重启之后必然被清空）。而为了快速恢复丢失的缓存数据，可以在节点重启后，利用 Memcached 命令行写个数据恢复脚本，然后将丢失的缓存数据从其他正常的 Memcached 服务器中恢复过来（每个节点上缓存有相同的数据），这样便可将丢失的缓存数据迅速恢复，从而降低其余缓存节点访问压力。要实现恢复功能很简单，只需从一个 Memcached 节点提取数据，然后将获取的数据写入要恢复的节点。

6.2 Redis 缓存系统

6.2.1 Redis 缓存概述

远程字典服务器（Remote Dictionary Server，Redis）是一款由 ANSI C 语言编写、遵守 BSD 协议、支持网络访问、基于系统内存的开源软件，同时，Redis 也是支持日志型和 Key-Value 键值对类型数据持久化的数据库。由于 Redis 中存储的 Value 值类型可以是字符串（String）、哈希（HASH）、列表（List）、集合（Sets）和有序集合（Sorted Sets）等类型，因此 Redis 通常也被称为数据结构服务器。在 Redis 中，这些数据类型都支持 Push/Pop、Add/Remove 以及针对集合的交集、并集和差集等丰富的数据操作，并且这些操作都具有保持数据一致性的原子性，在这些丰富的数据操作基础之上，Redis 还支持各种不同的数据排序方式。

与 Memcache 缓存系统类似，为了保证高效的数据存取效率，Redis 的数据都被缓存在内存中，但是 Redis 与 Memcache 在处理数据存储方式上有很大的差异，即 Redis 默认会周期性地 把缓存中更新的数据写入磁盘进行永久保存，并将对缓存数据的修改操作追加到日志文件中，由于 Redis 本质上就是一个基于内存的数据库，因此 Redis 具备很多如日志记录等通用数据库的共性。Redis 为了实现数据高可用，其利用已存入磁盘中的数据在不同节点之间进行同步操作，从而实现了 Master-Slave 模式的主从数据高可用，Redis 的数据可以从 Master 服务器向任意数量的 Slave 服务器上同步。除此之外，相对 Memcache 缓存，Redis 还提供了更多的语言支持，包括对 Java、C/C++/C#、PHP、JavaScript、Perl、Object-C、Python、Ruby 和 Erlang 等语言客户端 API 的支持。

6.2.2 Redis 数据交换

在 Redis 缓存系统中,并非所有通过 Redis 客户端写入的数据都永久性地存储在物理内存中,这是 Redis 和 Memcache 缓存系统最为明显的区别,Redis 会定期地将物理内存中的数据写入磁盘。由于相对磁盘存储,内存空间极其有限,为此,Redis 实现了用虚拟内存 (VM, Virtual Memory) 机制来扩充逻辑可用的内存空间,当 Redis 认为物理内存空间使用量达到某个预设值时,Redis 将自动进行内存页的交换 (Page Swap) 操作,从而将缓存中的数据写入磁盘中。通常在部署使用 Redis 时候,会根据系统实际物理内存的大小为其设置一个物理内存使用阈值,如果 Redis 发现物理内存的使用量超过了这个阈值,将会触发 Redis 的 Swap 操作,在数据交换过程中,Redis 会根据一定的算法 (如最少使用算法 LRU) 计算出哪些 Key 对应的 Value 需要交换到磁盘,之后就将这些 Key 对应的 Value 持久化到磁盘中,同时在内存中将其清除。

Redis 的 VM 机制使得 Redis 可以存储超过系统本身物理内存大小的数据,但是,这并不意味着可以无限减小物理内存并不断增加虚拟内存以满足 Redis 的整体内存需求,原因有两点:

其一,Redis 将全部的 Key 保存在物理内存中,从而可以快速命中 Key,并且由于 Redis 在读取数据时只在物理内存中进行 Key 的索引,找到 Key 之后再物理内存或者虚拟内存中读取对应的 Value,因此为 Redis 预留的物理内存空间至少要满足 Key 的存储。

其二,过度地减小为 Redis 预留的物理内存空间,会使得 Redis 频繁发生 Swap 操作,极大降低 Redis 的数据稳定性,即容易出现系统颠簸或者内存“抖动”,并对 Redis 的数据访问造成响应缓慢等影响。

此外,在 Redis 将内存中的数据交换到磁盘的时候,提供访问服务的 Redis 主线程和进行交换操作的子线程会共享这部分内存空间,所以如果此时有请求要更新这部分内存空间的数据,则 Redis 将会阻塞这个请求操作,直到子线程完成 Swap 操作后才可以进行数据的更新修改。

在 Redis 的数据交换过程中,由于 Redis 会将部分 Key 对应的 Value 通过 Swap 操作写入磁盘中,因此当命中的 Key 所对应的 Value 未存储在物理内存中时,Redis 就要通过 Swap 操作将数据从磁盘读回物理内存,然后再响应客户端请求。而在这种情况下,通常会出现 I/O 线程池的问题,默认情况下,Redis 会使用阻塞方式来使 I/O 请求等待 Swap 操作将全部所需的 Value 数据块读回内存,然后才进行下一步操作,Redis 的这种阻塞策略在少量客户端和批量操作情况下并无太大问题,但是在大型并发网站中,尤其是在大量并发随机小数据块请求的情况下,如果有很多命中 Key 对应的 Value 值都没有存储在物理内存中,则 Redis 的阻塞响应机制必然导致整个网站的访问请求极为缓慢,通常这种情况下必须设置 Redis 的运行 I/O 线程池,从而使得 Redis 并发进行磁盘到物理内存的数据读回操作,从而减少请求 I/O 的阻塞时间,提高网站的访问速率。

6.2.3 Redis 数据持久化

与 Memcache 缓存系统相比, Redis 最显著的功能特点之一便是数据持久化功能, 在 Redis 节点重新启动之后, 丢失的缓存数据会从磁盘重新加载或者从日志文件中重新恢复, 这一功能非常类似于商业数据库软件 (如 DB2 和 Oracle) 的日志恢复功能。针对两种不同的缓存数据恢复方式, Redis 提供了两种数据持久化的方式, 分别是 Redis 数据库方式 (RDB, Redis DataBase) 和日志文件方式 (AOF, Append Only File)。RDB 持久化方式就是将 Redis 缓存数据进行周期性的快照, 并将其写入磁盘等永久性存储介质中, 而 AOF 采用的是与 RDB 完全不一样的数据持久化方式, 在 AOF 方式下, Redis 会将所有对缓存数据库进行数据更新的指令全部记录到日志文件中, 待 Redis 重新启动时, Redis 将会把日志文件中的数据更新指令从头到尾重复执行一遍, 这是一个重现数据变更的过程, 当执行到日志文件末尾时即可实现缓存数据的恢复, AOF 恢复方式与 DB2 数据库的 redo 日志回滚恢复方式原理类似。

在实际的 Redis 使用过程中, RDB 和 AOF 这两种数据持久化方式可以同时使用, 并且这种方式也是 Redis 官方推荐的数据持久化方式, 在两种方式同时开启的情况下, 如果 Redis 数据库重新启动, 则会优先采用 AOF 方式来进行数据恢复, 这是因为 AOF 方式对数据恢复的完整性要高于 RDB 方式。另外, 如果用户认为 Redis 缓存数据无需进行持久化, 则可以将 RDB 和 AOF 数据持久化功能关闭。当 Redis 不进行任何的数据持久化操作时, Redis 就是一个基于内存的数据库, 此时的 Redis 在功能上和 Memcache 缓存系统并无太大差异。

在 Redis 的 RDB 持久化方式下, Redis 会将某一时刻的 Redis 数据库进行快照, 然后将快照数据写入到磁盘文件中进行持久化保存。由于 Redis 的 RDB 持久化操作会周期进行, 所以 Redis 在 RDB 数据持久化的操作过程中, 会先将数据库的快照数据写入到磁盘临时文件中, 待确定快照数据全部写入临时文件后, Redis 便会用这个临时文件覆写上一次 RDB 持久化生成的备份文件, 这样便可保证在 Redis 周期性的 RDB 数据持久化操作中, 磁盘文件系统中始终只保留一个最新的 Redis 缓存数据备份文件。在进行 RDB 数据持久化操作时, Redis 会单独创建 (Fork) 一个子进程来进行持久化操作, 而主进程是不会进行任何磁盘 I/O 操作的, 这样就确保了 Redis 在进行 RDB 操作的同时还保证了自身高效的请求响应。如果 Redis 缓存数据库较大, 在数据持久化和恢复时需要进行大规模数据的磁盘 I/O 操作, 同时对于数据恢复的完整性不是非常敏感, 那么 RDB 方式要比 AOF 方式更加高效。不过, RDB 持久化方式也有其自身的缺点, 例如, 如果对数据恢复的完整性非常敏感, 那么 RDB 方式就不能满足这种需求, 因为即使设置 RDB 持久化的时间间隔为 2 分钟, 当 Redis 故障恢复时, 也会有近 2 分钟的数据因未做持久化而丢失。这种情况下, 用户可以使用另一种持久化方式, 即 Redis 的 AOF 持久化方式。

AOF 是一种操作日志记录形式的持久化方式, 并且 AOF 的记录日志文件只允许往后追

加不允许改写已记录的条目。AOF持久化方式的原理,就是将Redis执行过的数据更新指令追加到日志文件中,并在数据恢复时按照先后顺序将指令都执行一遍,即再现一次数据的缓存过程,从而将数据恢复到故障点。AOF方式的开启很简单,只需将Redis配置文件Redis.conf中的appendonly设置为yes就可以打开Redis的AOF持久化功能,在开启AOF之后,如果对Redis缓存发起写请求(如SET命令),则Redis就会将SET命令作为一个新的记录追加到AOF文件的末尾。默认情况下,AOF持久化策略每秒进行一次日志文件的追加操作,并将缓存中的写指令记录到磁盘AOF文件中,因此,在默认AOF持久化情况下,当Redis故障恢复时,也仅有将近1秒钟的数据丢失。

此外,如果在追加日志时,恰好遇到磁盘空间满、文件系统inode耗尽后或突发断电等意外情况导致日志写入不完整,Redis还提供了Redis-check-aof工具进行AOF的日志修复。由于Redis采用了追加记录的方式写入同一个日志文件,因此,如果不对日志文件进行类似压缩归档的处理,则AOF文件必然会不断增加。为了应对不断增大的日志文件,Redis提供了AOF文件的重写(Rewrite)机制,即当AOF文件的大小超过预设阈值时,Redis就会启动AOF文件的内容压缩,在压缩过程中,仅保留恢复数据的最小指令集到一个临时AOF文件中,当压缩完成后,便会替换原有AOF文件。

Redis的两种持久化方式各有优缺点,用户可以选择其中一种进行Redis缓存数据的持久化,也可以同时使用RDB和AOF持久化数据,Redis官方推荐用户同时使用RDB和AOF。RDB的优点在于周期性批量进行持久化,恢复时只需将快照文件读入内存,因此恢复过程快速简单,但是由于存在快照周期,RDB方式很难保证数据恢复的完整性。与RDB相比,AOF方式在数据恢复的完整性上要高很多,但是在同等数据规模的情况下,AOF方式在数据恢复时,由于要重现日志文件中记录的全部操作,其恢复速度比RDB方式要慢很多。

6.2.4 Redis数据高可用

在Redis3.0版本发行之前,Redis自身并不具备集群高可用功能,尽管Redis3.0中增加了Redis的Cluster功能,但是,使用Redis Cluster功能的用户相对还是很少,其稳定性和可靠性仍需得到有力证明。就目前而言,针对Redis的集群高可用功能,更多的是借助第三方负载均衡和集群管理软件来实现或者是通过对Redis进行二次开发来实现。在Redis3.0的Cluster功能发布之前,Redis集群的数据高可用主要是通过Redis的主从复制(Master-SlaveReplication)功能来实现。在生产实践中,Redis高可用的具体实现方案有两种主流方式。

(1) Pacemaker或Keepalived集成

在多数部署情况下,Redis借助第三方软件负载均衡软件Keepalived或者集群管理软件Pacemaker来实现集群数据的高可用。大致实现过程是,通过Keepalived的VRRP协议或者创建Pacemaker的虚拟IP资源以对外提供高可用的Redis虚拟IP,并通过Pacemaker创

建 Redis 的 Master-Slave 多状态资源, 当 Redis 的 Master 节点故障后, 通过 Pacemaker 的资源管理功能, 利用 Redis 的 Agent 脚本将 Slave 节点提升为 Master, 同时虚拟 IP 迁移到新的 Master 节点继续对外提供服务。此类借助第三方软件来实现 Redis 集群高可用的配置方法, 其优点是主从节点的故障切换过程中, Redis 集群仍然对外提供固定不变的访问地址, 即切换过程对应用程序透明, 缺点是引入 Keepalived 或 Pacemaker 后, 使得原本简单配置即可使用的 Redis 集群, 由于第三方软件的引入而使得配置维护都更为复杂。

(2) Redis Sentinel 集群管理工具

Redis Sentinel 是 Redis 官方提供的集群管理工具, Sentinel 能持续监控主从节点 Redis 实例的工作运行状态, 当 Sentinel 监控到 Redis 实例出现故障时, Sentinel 将会通过相应的 API 通知系统管理员或其他应用程序。此外, 如果 Sentinel 监控到作为 Master 的 Redis 实例工作状态不正常, 则 Sentinel 集群管理工具将会启动故障恢复机制, 将一个处于 Slave 状态的 Redis 实例提升为 Master 实例, 而其他仍然处于 Slave 的实例将会自动更新其注册的 Master 节点名称, 同时应用程序会收到一个更换新的 Redis 访问地址的通知。

Redis Sentinel 本身是一个分布式系统, 用户可以部署多个 Sentinel 实例来监控同一组 Redis 实例, Sentinel 集群通过 Gossip 协议来确定一个 MasterRedis 实例是否故障, 并通过 Agreement 协议来执行故障恢复和配置变更, 一般在生产环境中部署多个 Sentinel 实例来提高系统可用性, 只要有一个 Sentinel 实例运行正常, 就能保证正常监控 Redis 实例的运行状态。使用 Redis Sentinel 进行 Redis 集群管理的不足之处, 就是 Redis 的主从切换对应用不透明, 即 Redis 主从节点切换后, 应用程序端无法自动识别新的 MasterRedis 地址, 因此应用程序端也得跟着变更地址。

Redis 主从模式的集群能够实现数据高可用的关键, 是 Redis 提供的数据 Replication 功能, 即数据能够在 Redis 的主从节点之间进行复制, 从而保证了数据的高可用性。Redis 的 Master 节点可以与一个或多个 Slave 节点组成 Redis 的主从模式集群, 当 Redis 的 Master 节点接收到 Slave 发出的数据同步请求后, Master 会对内存数据做一个快照, 并将快照数据存入 RDB 文件, 然后将 RDB 文件传送到 Slave 节点, Slave 节点接收到 RDB 文件后将其存入磁盘并重新加载到内存中, 以备后续成为 Master 节点时可以快速提供数据访问。Redis 的 Replication 功能具有如下特点:

- ❑ Redis 的 Replication 功能是异步复制的, 在 Redis2.8 版本之后, Slave 节点将会周期性地应答复制流中的数据量。
- ❑ Redis Master 可以同时连接多个 Slave 节点。
- ❑ Slave 节点又可以被其他 Slave 节点连接。
- ❑ 在复制的过程中, Master 节点是非阻塞的, 不会受到 Slave 节点数据同步的影响, 在 Slave 进行同步的时候, Master 继续提供高效的内存数据查询服务。
- ❑ 在复制的过程中, Slave 节点也是非阻塞的。在数据同步时, 是允许 Slave 节点利用之前的数据集提供查询功能, 还是此时返回一个错误, 并提示数据正在同步中,

用户可只通过 Slave 节点的 Redis.conf 文件来配置 Slave。但是不管怎么设置，在 Slave 同步完成之后，Slave 节点的陈旧数据将会被删除，新同步来的数据将会被加载到内存，而在这个删除与加载的过程中，Slave 的访问是暂时被禁止的。

- ❑ Redis 的 Replication 功能除了数据冗余之外，还可以将数据的只读查询分散到多个 Slave 节点，以让 Slave 节点来负责这类数据的排序操作。
- ❑ 使用 Redis 的 Replication 功能后，可以通过 Master 节点的 Redis.conf 将所有 save 语句都注释掉，这样 Master 就不会将数据写入本地磁盘，从而降低 Master 节点的 I/O 负载。与此同时，将 Master 节点数据全同步到 Slave 节点以进行异地保存，如果配置成这种工作方式，注意 Master 的 Redis 不能配置为自动重启。

6.2.5 Redis 高可用配置

如果用户配置了一个或多个 Slave 节点并将其连接到 Master 节点，则不管 Slave 节点是第一次连接 Master 还是重新连接到 Master，当连接完成之后，Slave 都会发送一个同步命令给 Master，Master 接到命令后开始在后台进行内存数据快照，并将在快照期间对当前缓存数据进行更新的命令全部缓存起来。当 Master 上的数据以 RDB 方式保存完成后，Master 将该 RDB 数据文件传送给 Slave，Slave 将数据文件写入本地磁盘，然后再将这些来自 Master 的 RDB 数据文件加载到内存中，之后 Master 将数据快照期间缓存的命令发送给 Slave，Slave 依次执行这些数据变更命令，使得每次同步操作后都能确保自身数据与 Master 节点缓存数据的一致性。

在 Redis 的 Master-Slave 集群模式下，当 Slave 与 Master 的连接因某些原因断开后，Slave 会自动与 Master 重新建立连接。如果 Master 同时收到多个 Slave 发送的数据同步请求，则 Master 仅会对缓存数据做一次快照保存，然后将保存的 RDB 数据文件发送给多个 Slave 节点。当同步中的 Slave 与 Master 断开又重新连接后，Slave 将会重新发送同步全部数据的请求，但是从 Redis 2.8 开始，重新连接的 Slave 也可以发送仅同步上次未完成同步的数据请求。Redis 的这种续传功能主要是在数据同步过程中，将同步操作的日志记录到内存文件中，Master 和所有的 Slave 还记录了数据同步的偏移量和 Master 此时的运行标志 (Run ID)，如果数据同步过程被断开，Slave 将会重新连接并请求 Master 继续同步。此时，如果 Master 的 Run ID 还没有改变，并且设置的同步偏移量在同步日志中可用，那么同步操作将会从中断点继续同步，如果不能满足这两个条件，则数据同步操作只能重新开始。此外，如果 Master 的 Run ID 没有保存到磁盘上，则无法进行续传同步。Redis 的主从数据同步方式分为 PSYNC 和 SYNC 命令同步方式，老版本 Redis 使用的是 SYNC 命令，Redis 2.8 以后，Redis 的 Slave 将会自动检测 Master 是否支持 PSYNC 命令，如果支持，则使用 PSYNC 命令同步方式，否则使用 SYNC 命令同步方式。

通常 Master 在接收到数据同步请求时，会将数据进行快照并保存到磁盘上的 RDB 文件中，然后同步到 Slave 节点，Slave 再将这些数据文件加载到本机内存中。如果 Master

节点的磁盘 I/O 性能较差，而不同的 Slave 节点不在同一时间点发生同步请求，则数据同步过程必然增加 Master 的工作负载。为了解决这个问题，Redis2.8.18 中提出了无盘复制 (Diskless Replication) 的设计，在这种模式下，Master 直接将 RDB 文件传送到 Slave 节点，而无须再在本机上进行 I/O 操作，无盘复制技术的提出极大提升了 Master 在数据同步复制过程中的响应效率。

Redis 在 Master-Slave 模式下的 Replication 功能配置是极为简单的，只需在 Slave 的配置文件 Redis.conf 中添加如下语句：

```
Slaveof Master_ip port
```

其中，Master_ip 是 Master 节点的 IP 地址，port 是 Redis 监听的端口号。如果 Master 需要一个密码验证，则可以在运行时给 Slave 设置密码或者通过配置文件永久性添加验证密码：

```
//运行时为Slave添加验证密码
config set Masterauth <password>
//配置文件中永久性添加验证密码
Masterauth <password>
```

从 Redis2.6 开始，Redis 默认 Slave 为 Read-only 模式，用户可以通过修改 /etc/Redis.conf 中的 Slave-read-only 为 false 来关闭该模式。在 Read-only 模式下，即使误操作也不会对 Slave 的数据进行修改，因为 Slave 禁止任何的写操作。但是，Slave 节点禁止任何写操作并不意味着用户可以把 Slave 节点暴露在互联网中，或者接收来自不信任客户端的访问，因为在该模式下，DEBUG 和 CONFIG 等管理员命令仍然可以在 Slave 上执行。为了详细介绍和解释 Redis 配置文件的功能参数，本节尽可能地对 Redis Replication 的功能配置文件 Redis.conf 进行参数描述，在 Master-Slave 模式下，Master 和 Slave 节点都需要 Redis.conf 配置文件，不过在 Slave 的 Redis.conf 文件中，除了需要为其指定 Master 节点外，Master 和 Slave 的 Redis.conf 配置并没有太大差异。Redis.conf 的配置文件按照参数功能可以划分为不同的配置段，即常规参数设置、RDB 持久化配置、Replication 数据主从同步配置、访问安全与内存限制配置、AOF 持久化配置以及高级配置等。

(1) 常规参数设置

Redis 中一个重要的参数就是配置 Redis 可以使用的物理内存空间大小，内存空间大小单位可以是 B、KB、MB、GB，而单位大小写是不敏感的，所以 1GB、1Gb 和 1gb 的写法都是完全一样的，内存单位转换关系如下：

```
1kb => 1024 bytes
1mb => 1024*1024 bytes
1gb => 1024*1024*1024 bytes
```

Redis 默认是不作为守护进程运行的，用户可以把 daemonize 参数设置为 "yes" 使其以守护进程来运行。当 Redis 作为守护进程运行时，Redis 会把进程 ID 写到 /var/run/Redis.

pid, 用户也可以通过 pidfile 参数修改进程文件的存放路径。

```
daemonize no
pidfile /var/run/Redis.pid
```

Redis 默认的监听端口是 6379, 如果端口设置为 0, 则 Redis 不会监听 TCP 套接字, bind 参数用以配置 Redis 监听 IP 地址, 如果未设置具体地址, 则所有接口的连接都会被监听。

```
port 6379
bind 127.0.0.1
```

unixsocket 用以指定监听连接的 unix 套接字的路径, 此参数没有默认值, 所以如果用户不显式指定, Redis 就不会通过 unix 套接字来监听请求, unixsocketperm 用以设置权限。

```
unixsocket /tmp/Redis.sock
unixsocketperm 755
```

timeout 参数用以设置客户端空闲多少秒后将其连接关闭, 0 代表禁用此功能, 即不管客户端是否活动, 都永不关闭其连接。

```
timeout 0
```

loglevel 参数用于设置 Redis 的运行日志, Redis 服务运行日志记录级别可能值有 debug、verbose、notice 和 warning, 其中 debug 会记录很多信息, 对开发/测试人员很有用, verbose 会产生很多精简有用信息, 但是不像 debug 等级那么多, notice 有适量的信息, 基本上是生产环境中需要记录的级别, warning 只有很重要或严重的信息会被记录下来。logfile 用于设置日志文件名, 也可以使用 "stdout" 来强制 Redis 把日志信息写到标准输出上。需要注意的是, 如果 Redis 以守护进程方式运行, 而用户设置日志显示到标准输出, 那么日志会发送到 /dev/null, 即不记录也不可见。用户可以配置系统日志记录器来记录 Redis 日志, 只需将 "syslog-enabled" 设置为 "yes", 然后根据需要设置相关的 syslog 参数即可。

```
loglevel verbose
logfile stdout
syslog-enabled yes
//指明syslog身份
syslog-ident Redis
//指明syslog设备, 必须是一个用户或者是 LOCAL0 ~ LOCAL7 外的设备
syslog-facility local0
//设置Redis数据库数目, 默认数据库是DB0, 可以通过SELECT <dbid> WHERE dbid (0~'databases'
- 1) 来为每个连接使用不同的数据库。
databases 16
```

(2) RDB 持久化设置

Redis 快照设置语法为 save <seconds><changes>, Redis 会在指定秒数 (seconds) 和数据变化次数 (changes) 之后把缓存数据库写到磁盘上。如果用户不需要进行 RDB 方式的数据持久化, 只需把所有 "save" 设置语句注释掉就行, 如果需要进行无盘 Replication, 则将

全部 save 注释掉。

```
//设置Redis在900s之后，并且存在100次以上数据变更，则对缓存数据库进行快照并写入磁盘RDB文件
save 900 100
```

用户可以通过参数 `rdbcompression` 选择在 RDB 数据持久化时，是否对 RDB 文件进行压缩，默认设置为 "yes"，即进行数据文件压缩，如果想节省 CPU 开销而不压缩数据，则可以把该参数设置为 "no"，但是非压缩情况下，数据文件就会很大，Replication 过程相对会变慢。此外，用户可以通过 `dbfilename` 参数设置保存数据的 RDB 文件名称，并通过 `dir` 参数设置存储 RDB 文件的系统目录。

```
rdbcompression yes
dbfilename dump.rdb
dir/dbbackup
```

(3) Replication 数据同步设置

Redis 的主从同步通过 `Slaveof` 配置来实现，此处的配置仅在 Slave 节点上操作，复制数据是指 Slave 本地从远端复制数据，本地可以有不同的数据库文件、绑定不同的 IP、监听不同的端口。如果 Master 设置了密码（通过 "requirepass" 参数来配置），那么 Slave 在开始同步之前必须进行身份验证，否则它的同步请求会被拒绝。当同步正在进行中，一个 Slave 与 Master 失去连接时，根据不同的 "Slave-serve-stale-data" 设置，Slave 的行为有两种可能：

如果 `Slave-serve-stale-data` 设置为 "yes"（默认值），则 Slave 会继续响应客户端请求，响应数据可能是正常数据，也可能是还没获得值的空数据；

如果 `Slave-serve-stale-data` 设置为 "no"，则 Slave 会以 "正在从 Master 同步 (YNC with Master in progress)" 来响应各种请求。Slave 可以根据指定的时间间隔向 Master 服务器发送 ping 请求，时间间隔可以通过 "repl_ping_Slave_period" 来设置，默认 10 秒。参数 "repl-timeout" 选项用于设置大块数据 I/O、向 Master 请求数据和 ping 响应的延长时间，默认值为 60 秒。需要注意的是，用户应该确保这个值比 `repl-ping-Slave-period` 大，否则 Master 和 Slave 之间的传输 timeout 时间比预期的要短。

```
//指定进行Replication的Master服务器
Slaveof <Masterip><Masterport>
//设置授权密码
Masterauth <Master-password>
//设置Replication中断后的Slave行为
Slave-serve-stale-data yes
//设置Slave ping Master主机的时间间隔
repl-ping-Slave-period 10
//设置timeout时间
repl-timeout 60
```

(4) 访问安全与内存使用设置

安全访问部分主要在 Master 节点上配置，此部分仅在要求处理任何客户端命令时，都

需要进行客户端身份和密码验证的情况下配置,这种配置在接收来自不可信客户端的命令时很有用。多数情况下,Redis 相关服务和应用程序都运行在安全区域内,尤其是 Slave 与 Master 之间的通信多数情况下并不需要身份验证。当然,如果确实有安全风险,则用户需要设置一个高强度的密码,由于 Redis 的响应速度太快,因此蓄意者可以在每秒内进行海量次数的密码破解尝试。在共享环境下,用户可以对相对比较危险的管理命令重命名,例如,可以将命令 CONFIG 改成其他人不容易猜到的名字,这样别人将无法使用该命令,也可以通过给命令赋值一个空字符串来完全禁用这条命令。

```
//设置Master的访问密码
requirepasspassword
//重命名CONFIG命令
rename-command CONFIG b840fc02d524045429941cc15f59e41cb7be6c52
//禁用CONFIG命令
rename-command CONFIG ""
```

用户可以通过“maxclients”参数设置允许同时并发连接的客户端数量,默认没有限制,该参数关系到 Redis 进程能够打开的文件描述符数量,特殊值“0”表示没有限制。一旦客户端连接达到这个限制,Redis 会关闭所有新连接并向客户端返回“达到最大用户数上限”(max number of clients reached)的错误信息。

```
//最大客户端连接数
maxclients 128
```

参数“maxmemory”用于设置分配给 Redis 使用的物理内存上限,在实际使用中,不要使用超过最大内存上限值。一旦内存使用达到上限,Redis 会根据选定的内存回收策略删除 Key。如果因为删除策略设置的问题,Redis 无法删除 Key,或者策略设置为“noeviction”,Redis 便会向客户端回复内存不足的消息。例如当内存不足时,执行 SET 和 LPUSH 等命令,便会收到类似信息,但是达到内存使用上限并不影响如 GET 等只读命令的执行。当多个 Slave 节点连接到已达内存使用上限的 Redis Master 实例时,响应 Slave 请求所需的输出缓存并不在“maxmemory”参数设置的内存当中,因此,建议用户不要设置过大的 Master 内存限制,以确保系统有足够的内存用作输出缓存。

```
//内存使用上限
maxmemory 1GB
```

如果达到内存限制,Redis 将根据设置的策略算法来确定如何删除 Key。Redis 支持的策略算法有下面五种。

- ❑ Volatile-LRU: 根据 LRU 算法生成的过期时间来删除 Key。
- ❑ AllKeys-LRU: 根据 LRU 算法删除任何 Key。
- ❑ Volatile-random: 根据过期设置来随机删除 Key。
- ❑ AllKeys-random: 无差别随机删 Key。
- ❑ Volatile-TTL: 根据最近过期时间来删除。

❑ Noeviction：不删除任何 Key，直接在写操作时返回内存不足的错误。

对所有策略来说，如果 Redis 找不到合适的 Key 来对其删除，Redis 就会在写操作时返回内存不足的错误信息。对 LRU 和最小 TTL 算法而言，其实现精度都不是很高，但是也比较符合实际需求，用户可以通过设置 LRU 和 TTL 的采样样本来确定删除哪个 Key，Redis 默认会取三个 Key 进行对比，然后删除最旧的那个 Key。

```
//达到内存上限时的Key删除策略
maxmemory-policy volatile-lru
//Redis Key采样样本数目设置
maxmemory-samples 3
```

(5) AOF 持久化配置

默认情况下，Redis 以异步形式把数据导出到磁盘上保存（RDB 持久化）。这种情况下，当 Redis 挂掉后，最新的数据就丢了，即 RDB 的数据恢复很难保证完整性。如果不希望丢掉任何一条 Redis 缓存数据，用户可以选择纯累加模式（AOF 持久化），一旦开启 AOF 模式，Redis 便会把每次写数据的操作都记录到 appendonly.aof 文件。每次启动时 Redis 都会把这个文件的数据读入内存并重新执行，以将数据恢复到最新点。RDB 导出的数据库文件和 AOF 纯累加文件可以并存，如果 RDB 和 AOF 模式同时开启，那么 Redis 会在启动时载入日志文件而忽略导出的 dump.rdb 文件。

```
//开启AOF模式
appendonlyyes
//纯累加文件名字，默认为appendonly.aof
appendfilename append_only.aof
```

在 AOF 模式下，Redis 支持三种不同的日志记录模式，分别是 always 模式、everysec 模式和 no 模式。其中，最安全的是 always 模式，每次写操作都立刻记录到 aof 文件，但是 always 模式也是最慢。no 模式表示不要立刻记录到 aof 文件，只有在操作系统需要记录的时候才写入 aof 文件，no 模式比较快，但是不安全。everysec 模式不是每秒写入一次，是针对 no 和 always 模式的折中方案，其能在速度和数据安全性之间取得比较好的平衡。设置 no 模式可以获得更好的性能表现，但是数据恢复不完整，设置 always 模式便会牺牲速度，但是确保了数据的安全性和完整性。如果 Redis 的 AOF 记录模式设置为 always 和 everysec 模式，则会在系统上产生大量 I/O 进程，过多的 I/O 进程极有可能阻塞对 Redis 的请求访问。不过，目前仍然没有很好的方式来处理这个问题，一个相对安全的办法，就是设置“no-appendfsync-on-rewrite”参数为“yes”，使得 Redis 有更高的优先级来处理客户端的访问请求而不是忙于频繁的记录日志。当然，如果系统资源本身很强大，则将其设为“no”以保证 Redis 记录实时日志。

```
//AOF日志记录模式
appendfsync everysec
//AOF日志记录优先级设置
no-appendfsync-on-rewrite no
```

如果 AOF 日志文件达到指定的百分比, Redis 将通过 BGREWRITEAOF 自动重写 AOF 日志文件。用户还需要指定允许重写的日志文件的最小尺寸, 以免虽然达到了约定百分比, 文件尺寸仍然很小的情况也要重写日志文件, 指定百分比为 0 会禁用 AOF 的自动重写功能。

```
//达到100%才自动重写日志文件
auto-aof-rewrite-percentage 100
//日志文件至少64MB以上才允许重写
auto-aof-rewrite-min-size 64mb
```

(6) 虚拟内存与数据交换设置

虚拟内存可以使 Redis 在物理内存不足的情况下仍然可以将所有数据序列保存在内存里。为了做到这一点, 高频 Key 会调到内存里, 而低频 Key 会转到交换文件里, 就像操作系统使用内存页一样。要使用虚拟内存, 只要把“vm-enabled”设置为“yes”, 并根据需要设置虚拟内存参数就即可。

```
//启用虚拟内存
vm-enabled yes
```

用户可以为交换文件指定路径, 但是交换文件不能在多个 Redis 实例之间共享, 所以确保每个 Redis 实例使用一个独立交换文件, 对于交换文件而言, 最好的保存交换文件的介质是固态硬盘 SSD。

```
//Redis交换文件路径
vm-swap-file /tmp/Redis.swap
```

参数“vm-max-memory”用于配置虚拟内存可用的最大容量。在启动了虚拟内存功能后, 只要交换文件还有空间, 物理内存达到使用上限时数据便会交换到磁盘上的交换文件里。“vm-max-memory”设置为 0 表示系统在用掉所有可用物理内存后才发生数据交换, 通常设置为使用率超过 60%~80% 便发生内存交换比较合理。数据在内存中是按页存储的, 一个存储对象可以存储在多个连续页里, 但是一个数据页无法被多个存储对象共享。所以, 如果数据页太大, 那么小对象就会浪费掉很多空间, 如果数据页太小, 交换过程就会过于频繁。因此, 如果数据是很多小对象, 建议 page size 为 64 字节或 32 字节, 如果使用很多大对象, 那就用更大一些的尺寸, 如果不确定, 那就用默认值。另外还需设置的虚拟内存参数是页数目 vm-pages, 其值等于交换文件大小除以页大小。此外, 开启的虚拟内存 I/O 线程数也是很关键的参数, I/O 线程可以完成从交换文件读写数据到内存的操作, 也可以处理数据在内存与磁盘间的交互和编码/解码的操作, 适当多开启一些线程, 在一定程度上可以提高处理效率。

```
//物理内存使用率超过70%发生内存交换
vm-max-memory 70
//页大小
vm-page-size 32
//交换页数目
vm-pages 134217728
```



```
//可同时运行的虚拟内存I/O线程数
vm-max-threads 4
```

(7) 高级配置

当有大量数据时，可以使用哈希编码（需要更多的内存），不过元素数量上限不能超过给定限制，用户可以通过下面的选项来设定这些限制：

```
HASH-max-ziplist-entries 512
HASH-max-ziplist-Value 64
```

与哈希相类似，在数据元素较少的情况下，可以用另一种方式来编码，从而节省大量内存空间，而这种方式只有在符合下面限制的时候才可以用：

```
list-max-ziplist-entries 512
list-max-ziplist-Value 64
```

还有另外一种特殊的编码情况，即数据全是由 64 位无符号整型数字构成的字符串，下面的配置项就是用来限制该情况下使用这种编码的最大上限：

```
set-max-intset-entries 512
```

使用哈希编码时，存在哈希刷新，即每 100 个 CPU 毫秒会拿出 1 毫秒来刷新 Redis 的主哈希表（顶级键值映射表）。Redis 所用的哈希表实现采用延迟哈希刷新机制，即用户对一个哈希表操作越多，哈希刷新操作就越频繁；反之，如果系统非常不活跃那么也就是占用点内存保存哈希表而已。默认是每秒钟进行 10 次哈希表刷新，用来刷新字典，然后尽快释放内存。如果对延迟比较在意，就设置“ActivereHASHing”为“no”，如果不太在意延迟而希望尽快释放内存，就设置“ActivereHASHing”为“yes”。

```
ActivereHASHing yes
```

在用户有标准配置模板但是每个 Redis 服务器又需要个性设置时，可以使用 Redis 配置文件支持的包含一个或多个其他配置文件功能，包含文件特性允许用户引入其他配置文件。

```
include /path/to/local.conf
include /path/to/other.conf
```

6.2.6 Redis 集群概述

在 Redis 3.0 发布之前，Redis 并没有官方的 Cluster 可用。在这之前，RedisCluster 用得最多的应该是 twitter 发布的 Twemproxy(<https://github.com/twitter/twemproxy>) 和豌豆荚开发的 codis(<https://github.com/wandoulabs/codis>)。Redis 在 3.0 版正式引入了集群功能，Redis 集群是一个分布式（Distributed）、容错（Fault-Tolerant）的 Redis 内存 Key-Value 存储实现，Redis 集群功能是普通 Redis 节点的功能子集，Redis Cluster 是 Redis 的分布式实现，RedisCluster 在设计之初就考虑到了去中心化的问题，因此集群中不存在任何中心控制节点，每一个节点都是功能对等的集群成员，每个节点除了保存有自己的配置元数据外，

还保存着整个集群的状态信息。由于每个集群节点之间彼此互联，并且随时保持活动连接，客户端只需连接到集群中的任一个节点，便可获取到存储在其他节点上的数据，Redis Cluster 的架构拓扑如图 6-7 所示。

Redis Cluster 中的各个节点之间保持相互连接，并且彼此之间可以通信，客户端随意连接到任何一个集群节点就能将整个 Redis 集群作为一个整体来访问，同时客户端也无需知道 Redis Cluster 将其提交的数据存入哪个 Redis 节点中，数据存储完全由 Redis Cluster 根据自己的算法来决定。

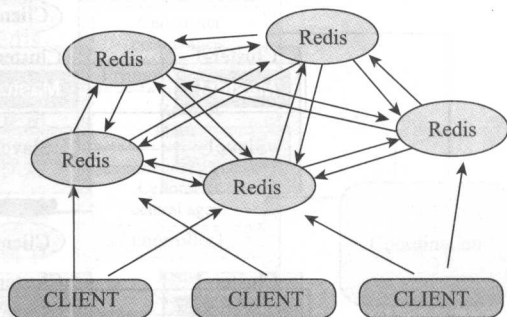


图 6-7 RedisCluster 架构图

在 Redis Cluster 中，数据被分散存储到不同的 Redis 节点上，Redis Cluster 并没有采用一致性 HASH 来分配数据存放到不同的节点，而是采用了一种称为哈希槽（HASH Slot）的技术来分配数据。Redis Cluster 默认将存储空间分配为 16384 Slots，每个节点承担其中的一部分 Slots。假设现在有三个 Redis 节点，三个 Redis 节点可以是三个独立的操作系统，也可以是同一个系统上不同的端口号，现在将 16384 个 Slots 分配到三个节点上，那么每个 Redis 节点上分配到的 Slots 可以如下：

□ 0-5460 号槽位分配到节点 A。

□ 5462-10922 号槽位分配到节点 B。

□ 10923-16383 号槽位分配到节点 C。

如果 Redis 需要扩充 Redis 节点数，只需重新分配每个节点的槽位。当客户端通过 SET 命令来保存一个 Key-Value 键值的时候，Redis Cluster 会采用 CRC16 (Key) 对 16384 取模来决定该 Key 应该被存储到哪个 Slots 中，具体的计算公式就为： $\text{slot_num} = \text{CRC}(\text{'Key_name'}) \% 16384$ 。假设通过公式计算得到的 Slot_num 为 14201，那么此 'Key_name' 就会被存储到位于 C 节点上的槽位中，当客户端连接到 A、B、C 中任一节点提取 'Key_name' 的值时，Redis Cluster 也会采用同样的算法，计算出存储该 Key 的节点和槽位，然后到 C 节点中的 14201 Slot 提取数据。

为了保证数据的高可用，Redis Cluster 用到了 Master-Slave 的主从数据复制模式，即为每个 Master 都设置一个或多个 Slave，Master 负责数据存取，而 Slave 负责数据的同步复制，当 Master 故障时，其中的某个 Slave 会被提升为 Master。在上例的 A、B、C 三个节点中，由于没有为节点设置 Slave，假如 B 节点故障，则位于 B 节点上的数据将无法访问，因此从高可用角度考虑，需要在 Redis Cluster 中设置 Master 节点和 Slave 节点，假设集群中有 A、B、C 三个主节点，同时分别为其设置了 A1、B1、C1 三个 Slave 节点，此时如果 B 节点故障，则集群会将 B1 节点选取为 Master 节点继续提供服务，当 B 节点恢复之后，它就变成了 B1 的 Slave 节点。当然，如果 B 和 B1 同时故障，则集群就不可用了，Redis Cluster

的集群 Master 节点选举机制如图 6-8 所示。

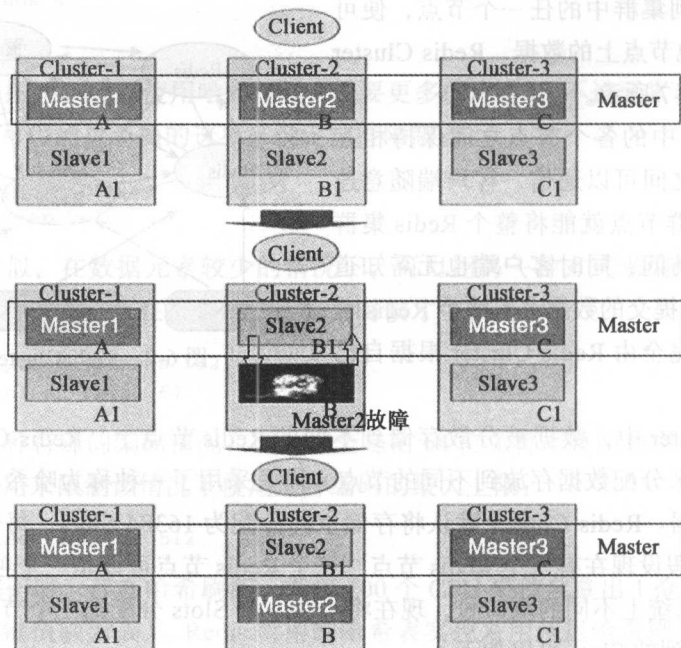


图 6-8 Redis Cluster 节点故障切换过程

关于 Redis Cluster 的详细配置和更多功能参数的介绍, 请参考 Redis 官方网站关于 RedisCluster 的描述^①, 感兴趣的读者可以自行参考。

6.2.7 Redis 在 OpenStack 中的应用

在 OpenStack 开源云的计费项目 Ceilometer 中, 当进行规模化集群部署时, 通常使用 Redis 插件来为运行在控制节点集群上的多个 Ceilometer Agents 提供协调机制, Redis 插件使用具有 Redis 后端的 Tooz 库来为 Agents 提供一组轮询使用的资源集。在部署了 Redis 插件之后, Ceilometer Agents 的部署可以分布到每个控制节点上, 并且这些分布的 Agents 将会自动加入协调组 (Coordination Group) 中, 在这种分布式集群部署中, 通常使用 Pacemaker 创建 Redis-server 资源以监控 Redis 插件进程的运行状态, 同时插件会自动配置 Redis-sentinel 进程来监控 Redis 集群状态, 而 Redis-sentinel 的主要作用是当 Redis 集群的 Master 节点故障时, 通过一定的机制重新选取新的 Master 节点, 同时将 Ceilometer 的代理重新定向到新的 Master 节点, 并进行 Redis 集群节点之间的数据同步。总体而言, 在 OpenStack 高可用集群部署中, 使用到 Redis 功能的组件主要是 Ceilometer, Ceilometer 的各个分布式 Agents 借助了 Redis 的功能来实现了高可用部署。

① <http://Redis.io/topics/Cluster-tutorial> 和 <http://Redis.io/topics/Cluster-spec>

在 OpenStack 集群高可用部署中, Redis 被用作 Ceilometer 组件的分布式协调组后端缓存, 同时集合集群资源管理软件 Pacemaker, 将 Redis 配置为 Pacemaker 的资源以实现 Redis-server 的高可用 Master-Slave 模式。此外, 还需要为 Redis 集群配置一个高可用 IP 地址, 高可用 IP 可以通过 Pacemaker 的虚拟 IP 资源实现。在三节点的 OpenStack 控制节点集群架构中, 将其中一个控制节点作为 Redis 的 Master 节点, 另外两个控制节点作为 Redis 的 Slave 节点, Redis 和 Ceilometer 的 Central agent 在三个控制节点中的部署拓扑如图 6-9 所示。

在集成 Pacemaker 的集群中, Redis 服务被配置成为 Pacemaker 的多状态资源, 即 Master/Slave 资源组, 假设三个控制节点分别为 controller1-vm、controller2-vm 和 controller3-vm, 同时集群里有两个计算节点 compute1 和 compute2, 则正常情况下, Redis 集群资源组的运行状态应当如下:

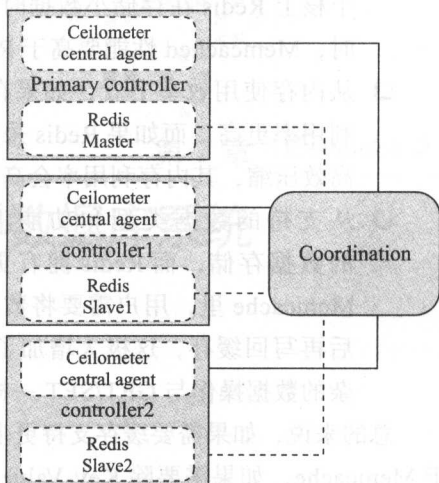


图 6-9 OpenStack 集群中 Ceilometer 与 Redis 部署架构

```
Master/Slave Set: Redis-server-Master [Redis-server]
Masters: [ controller2-vm ]
Slave: [ controller1-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
vip-Redis      (ocf::heartbeat:IPaddr2):      Started controller2-vm
```

其中, Redis-server-Master 为多状态 Redis 资源组名称, vip-Redis 为 Redis 对外服务的虚拟 IP 资源。从 Pacemaker 的资源状态中可以看到, 通过 Pacemaker 配置的 Redis 高可用 Master/Slave 集群中, controller2-vm 为 Redis 的 Master 节点, controller1-vm 和 controller3-vm 为 Redis 的 Slave 节点, 同时 Redis 对外提供服务的虚拟 IP 地址运行在 Master 节点上, 这是通过 Pacemaker 的 Colocation 约束设置的, 即 Redis 的 VIP 只会运行在 Master 节点上。如果 Redis 的 Master 节点故障, 则 Pacemaker 将会从两个 Slave 节点中重新选举一个 Master 节点, 而 VIP 也会相应地自动移动。

6.3 本章小结

本章讲述了目前使用极为广泛的 Memcache 与 Redis 缓存系统, 并着重讲解了二者在集群配置和数据高可用方面的配置和使用方法。在 OpenStack 高可用集群部署过程中, 通常使用 Memcache 和 Redis 来频繁访问关键数据的缓存, 以提高用户请求的响应速率并减少后端 MySQL 数据库的访问压力。在实际使用过程中, Memcache 与 Redis 在一定程度上可以互换使用, 二者都是基于内存的缓存数据库, 但是两者仍然存在一些使用上的差别。

- ❑ 从性能上看, 由于 Redis 是单核运行, 而 Memcached 可以使用多核, 所以平均每一个核上 Redis 在存储小数据时比 Memcached 性能更高, 但在存储 100k 以上的数据时, Memcached 性能要高于 Redis。
- ❑ 从内存使用效率上看, 如果存储简单的 Key-Value 键值对, 则 Memcached 的内存利用率更高, 而如果 Redis 采用 HASH 结构来做 Key-Value 存储, 由于其组合式的高效压缩, 其内存利用率会高于 Memcache, 但是会占用一定的 CPU 负载。
- ❑ 从支持的数据类型和功能上看, Memcache 通常仅支持 Key-Value 键值对格式的数据存储, 而 Redis 拥有更多的数据结构并支持更丰富的数据操作。通常在 Memcache 里, 用户需要将数据拿到客户端来进行类似 Redis 支持功能的修改, 然后再写回缓存, 这极大增加了网络 I/O 的次数和数据体积。而在 Redis 中, 这些复杂的数据操作与 GET/SET 一样高效。

总的来说, 如果需要缓存支持更多更复杂的数据结构和数据操作, 那么选用 Redis 会优于 Memcache, 如果需要除 Key-Value 之外的更多数据类型支持, 使用 Redis 更合适, 如果需要缓存海量大数据, 并且数据很关键, 则使用 Redis 会更高效和安全。而如果仅需要存储简单的 Key-Value 键值对数据, 则选用 Memcache 能更好地利用内存, 如果存储的数据对象较大, 则使用 Memcache 可以得到更高的性能。

集群数据库系统

数据库是 Openstack 集群中最为核心的组件之一，为整个 Openstack 云操作平台提供了完善的数据存储和查询服务，Openstack 各个服务组件产生的状态数据和配置数据均会存放到数据库中。在开源数据库领域，存在诸多数据库产品，按照数据库的数据存储形式又可以分为 SQL 关系型数据库和 NoSQL 非关系型数据库，其中关系型数据库的代表有 MySQL、MariaDB、PostgreSQL、SQLite 和 OcenBase 等，而 NoSQL 类型数据库的代表有 MongoDB、Cassandra、CouchDB、Hadoop HBase、MemcachedB 和 RedisDB 等。在 Openstack 官方给出的参考部署中，关系型数据库使用的是 MySQL 的主分支数据库 MariaDB，而非关系型数据库使用的是被广泛使用和认可的 MongoDB 数据库。本章将重点介绍 MariaDB 和 MongoDB 两款数据库的安装配置和高可用数据库集群的部署，此外，本章高可用数据库集群的部署是后续 Openstack 高可用集群部署的基础，一旦数据库不可用，则整个 Openstack 集群将处于不可用状态，因此，本章将重点讲解 MariaDB 和 MongoDB 数据库的高可用配置部分，关于两个数据库的具体命令及使用方式，读者可以自行参考相应的官方网站。

7.1 关系型数据库——MariaDB

7.1.1 MySQL 概述

MySQL 最初由瑞典 MySQLAB 公司开发，是一款小型关系型数据库管理软件，其最初的开发者为 MontyWidenius 等人。MySQLAB 公司在 2008 年 1 月被 SUN 公司收购，而 SUN 公司于 2009 年被 Oracle 收购，因此目前 MySQL 的商业版本由 Oracle 授权发行。尽

管 MySQL 受到 Oracle 收购事件的影响,但是由于其体积小、速度快、总体拥有成本低等特点,尤其是 MySQL 开放源码的特性,使其在各种中小型 Internet 网站中仍然极具人气,许多互联网创业公司为了降低软件成本,均选择 MySQL 作为网站后端数据库。就 Openstack 而言,社区在最初发行的 Openstack 版本中,官方指定的各个项目后端关系型数据库也为 MySQL,但是到了目前的稳定版本,如 Juno, Kilo, Liberty 和 Mitaka 等,都已将 MariaDB 作为 Openstack 官方指定的后端关系型数据库。当然, MariaDB 作为 MySQL 的主分支,其在使用上与 MySQL 并无明显差别。

同很多主流的大型商业数据库,如 Oracle、DB2、SQL Server 相比,MySQL 也存在自身的不足之处,例如规模小、功能有限等,但是在过去除了像金融和电信等行业很少使用 MySQL 外,由于其开源、简单和易用等特点,MySQL 在数据库领域的知名度丝毫不亚于这些企业级重量数据库。对于个人、小型单位组织和中小型企业来说,MySQL 提供的功能已经足以满足各种数据管理的需求,加上 MySQL 源码开放带来的自主可控和根据自身需求可进行二次开发的特性,MySQL 已成为目前互联网领域流行门户网站架构的四套件(Linux+Apache+MySQL+PHP)之一,即采用 Linux 作为操作系统,使用 Apache 作为 Web 服务器,以 MySQL 作为网站后端数据库,并用 PHP 作为脚本解释器,这种架构配置已成为很多中小型网站的首选方案。

MySQL 能够在互联网行业和开源社区普及壮大,并在个人和很多中小型企业中得到普遍使用和认可,与其体积小但功能强大的特性是密不可分的,这也是它在各种操作系统和行业中极受欢迎的原因,MySQL 的功能特性归结起来有以下几方面:

- ❑ 使用 C 和 C++ 编写,并使用了多种编译器进行测试,保证源代码的可移植性。
- ❑ 支持 AIX、FreeBSD、HP-UX、Linux、Mac OS、Novell Netware、OpenBSD、OS/2 Wrap、Solaris、Windows 等多种操作系统。
- ❑ 为多种编程语言提供了 API 接口,这些编程语言包括 C、C++、Python、Java、Perl、PHP、Eiffel、Ruby 和 Tcl 等。
- ❑ 支持多线程,充分利用 CPU 资源。
- ❑ 优化的 SQL 查询算法,有效地提高查询速度。
- ❑ 既能够作为一个单独的应用程序应用在客户端服务器网络环境中,也能够作为一个库嵌入到其他软件中提供多语言支持。
- ❑ 提供 TCP/IP、ODBC 和 JDBC 等多种数据库连接途径。
- ❑ 提供用于管理、检查、优化数据库操作的管理工具。
- ❑ 可以处理拥有上千万条记录的大型数据库。

Oracle 收购 SUN 之后,重新对 MySQL 的版本进行了划分,分成了社区版和企业版,企业版需要收取相关的许可费用,同时也会提供更多的功能。但是,Oracle 收购 MySQL 之后,MySQL 社区版也变得日渐封闭和更新缓慢,出于 Oracle 日后完全控制 MySQL 的担心,社区重新开启了一个 MySQL 的分支以避免被 Oracle 完全控制,这个分支便是让 MySQL 面

临严峻挑战的 MariaDB。从目前 MySQL 和 MariaDB 的发展和使用情况来看, MySQL 的前景几乎不被看好, 由于不满 Oracle 收购 MySQL 后对社区的控制行为, 目前众多 Linux 发行版本逐渐抛弃了这个曾经的人气开源数据库, 转而转向了 MariaDB。其中, 目前已经抛弃 MySQL 而选用 MariaDB 作为默认数据库的 Linux 发行版本至少有以下几种:

- ❑ Fedora Project。Fedora Project 的 Linux 发行版本 Fedora19 中已经改用 MariaDB 作为系统默认数据库。
- ❑ Slackware Linux。作为最古老的 Linux 发行版, Slackware Linux 认为 MariaDB 社区更有活力也更愿意和开源社区合作, 因此也在 2013 年 3 月宣布使用 MariaDB 替代 MySQL。
- ❑ Arch Linux。Arch Linux 于 2013 年 3 月宣布使用 MariaDB 替换 MySQL。
- ❑ Red Hat。Red Hat 于 2013 年 6 月宣布企业发行版 RHEL7 将不再使用 MySQL, 而是采用 MariaDB 替代 MySQL。而在同年的早些时候, Red Hat 的社区发行版 Fedora 已宣布从 MySQL 切换到 MariaDB, 对应 RHEL 版本的 Centos 自然也抛弃了 MySQL。

当然, 选择抛弃 MySQL 的 Linux 发行版本不只以上这些, 如 openSUSE 等也都从 MySQL 切换到了 MariaDB。随着 MariaDB 得到更多的认可, 类似的替换在更多的 Linux 发行版中已成为必然趋势。除了在 Linux 发行版本中被抛弃, MySQL 在很多重量级的互联网巨头中也不断被替换:

- ❑ Apple 公司。作为市值最大的科技公司, Apple 是最早抛弃 MySQL 的公司之一。在 Oracle 于 2009 年收购 SUN 之后, Apple 就抛弃了 MySQL, 并选择了另一种目前发展也很活跃的数据库 PostgreSQL。
- ❑ Wikipedia 网站。从 2012 年开始, 维基百科便着手准备从 MySQL 切换到 MariaDB 的测试, 而到目前为止, Wikipedia 已基本完成 MySQL 到 MariaDB 的迁移。
- ❑ Google 公司。作为互联网巨头的代表, Google、Facebook、Twitter 都大量使用过 MySQL, 然而 Oracle 收购 MySQL 后, Google 也开始致力于 MySQL 到 MariaDB 的迁移, 从而远离这个 Oracle 掌控下的开源数据库。
- ❑ Openstack 社区。作为最大的开源云计算社区, Openstack 官方推荐的各个开源项目后端数据库也从 MySQL 替换为 MariaDB。

随着开源技术的成熟和发展, 开源软件因为汲取了社区智慧和用户诉求而被广大用户普遍认可和接受, 相比商业软件的一家独大和垄断控制, 用户更喜欢可以进行二次开发而适应自身需求的开源软件, 而不是强迫自己的应用和产品去适应垄断的商业软件, 同时, 开源软件的使用者可以通过提交 BUG 的形式向社区反馈自己遇到的问题, 或者向社区提交 BUG 修复补丁, 这种社区互动的发展形式要远远优于商业软件由上向下的定期单向垄断发布。基于这种现象, MySQL 被商业软件公司 Oracle 收购后不断面临其他开源数据库的挑战也是必然结果, 而作为 MySQL 主分支的 MariaDB 以其真正开源的社区发行方式却受到越

来越多的用户青睐，并有取代 MySQL 之势也在情理之中。

7.1.2 MariaDB 概述

MariaDB (<http://MariaDB.org>) 是由 MySQL 创始人之一的 Monty Widenius 开创的一个 MySQL 分支版本，MariaDB 基于 GPL 2.0 发布，是一个完全开源的数据库。在 MySQL 数据库被 Oracle 公司收购后，Monty 担心 MySQL 数据库发展的未来，从而分支出一个版本并取名为 MariaDB，之后 MariaDB 便得到了大多数 Linux 发行版本和原 MySQL 用户的支持。MariaDB 和其他 MySQL 分支有很大的不同，其默认使用崭新的 Maria 存储引擎，是原 MyISAM 存储引擎的升级版本。此外，其增加了对 Hash Join 的支持和对 Semi Join 的优化，使 MariaDB 在复杂的分析型 SQL 语句中较原版本的 MySQL 性能提高很多。另外，除了包含原有的一些存储引擎，如 InnoDB、Memory，还整合了 PBXT、FederatedX 存储引擎。从目前开源数据库的使用情况来看，MariaDB 数据库是目前 MySQL 分支版本中性能最好和使用者最多的一个版本，尤其是在 OLAP 的应用中，对 Hash Join 的支持和对 Semi Join 的优化大大提高了 MariaDB 数据库在 OLAP 应用中的查询性能。

MariaDB 与 MySQL 在很多方面都是兼容的，对于开发者来说，使用起来几乎感觉不到任何不同。目前 MariaDB 是发展最快的 MySQL 分支版本，新版本发布速度已经超过了 Oracle 官方的 MySQL 版本。Monty 开创 MariaDB 分支的目的不仅仅是要替代 MySQL，其初衷更多是通过社区的努力提高 MySQL 的技术和性能，从而更好地服务于社区和用户。例如 Wikipedia 就曾宣称将英文维基的一个数据库从 MySQL 5.1 迁移到 MariaDB 5.5.28 后，通过全面测试发现，MariaDB 的查询效率提升了 3%~15%，平均提升了 8%，而且没有任何异常发生，吞吐量提升了 2%~10%，虽然没有具体数据来支撑这个测试结论，但是至少说明 MariaDB 在性能上是可靠的，而目前 MariaDB 的稳定发行版本为 MariaDB 10.1.14，新版本中还增加了 GIS 和 JSON 等新功能，相信随着社区的发展和壮大，新版本的 MariaDB 在可用性和可靠性方面都已今非昔比。同 MySQL 相比，MariaDB 具有以下优势：

- ❑ 可免费商业使用，MySQL 则有社区版和企业版之分。
- ❑ Maria 存储引擎支持。
- ❑ PBXT 存储引擎支持。
- ❑ XtraDB 存储引擎支持。
- ❑ FederatedX 存储引擎支持。
- ❑ 更快的复制查询处理。
- ❑ 线程池功能。
- ❑ 运行速度更快。
- ❑ 更多的扩展功能模块。
- ❑ 支持 Unicode 排序。

由于 MariaDB 与 MySQL 的完全兼容性，不仅包括 API 的兼容性，还包括客户端

协议等的兼容性,因此,由MySQL替换成MariaDB并不存在太多技术上的难题,根据MariaDB的技术文档,MySQL替换成MariaDB就像软件升级一样,如果是相同的基础版本,直接卸载MySQL然后安装MariaDB即可,如果MariaDB主版本较高,则只需安装之后再对其进行升级即可。

7.1.3 MariaDB 安装配置

目前很多新发行的Linux版本中都采用了MariaDB作为默认数据库,如RHEL7和CentOS7以及Fedora22等Linux发行版本中,MariaDB的安装包都已经集成到了系统ISO镜像中,用户在系统安装时即可选择安装MariaDB,如果系统安装时没有选择安装MariaDB,则可以通过MariaDB官方网站提供的安装包进行安装。MariaDB官方基金会提供了一个生成MariaDB安装包Yum库的网站,其网站地址为<https://downloads.MariaDB.org/MariaDB/repositories>。用户进入该网站后,可以根据自己的Linux系统版本进行选择,该网站会为你的系统配置生成一个安装包源文件,用户只需将该网站生成的安装源拷贝到系统的Yum源配置中,即可进行在线的MariaDB安装。例如使用CentOS7系统安装MariaDB10.1稳定版本,则在该网站上的配置如图7-1所示。

Downloads

Setting up MariaDB Repositories

To generate the entries select an item from each of the boxes below. Once an item is selected in each box, your customized repository configuration will appear below.

1. Choose a Distro

- openSUSE
- Arch Linux
- Mageia
- Fedora
- CentOS**
- RedHat
- Mint
- Ubuntu
- Debian

2. Choose a Release

- CentOS 7 (64-bit)**
- CentOS 6 (64 bit)
- CentOS 6 (32 bit)
- CentOS 5 (64 bit)
- CentOS 5 (32 bit)

3. Choose a Version

- 10.1 [Stable]**
- 10.2 [Alpha]
- 10.0 [Stable]
- 5.5 [Stable]

Here is your custom MariaDB YUM repository entry for CentOS. Copy and paste it into a file under /etc/yum.repos.d/ (we suggest naming the file MariaDB.repo or something similar).

```
# MariaDB 10.1 CentOS repository list - created 2016-08-28 03:52 UTC
# http://downloads.mariadb.org/mariadb/repositories/
[mariadb]
name = MariaDB
baseurl = http://yum.mariadb.org/10.1/centos7-amd64
gpgkey=https://yum.mariadb.org/RPM-GPG-KEY-MariaDB
gpgcheck=1
```

Yum 源配置语句

After the file is in place, install MariaDB with:

```
sudo yum install MariaDB-server MariaDB-client
```

安装语句

If you haven't already accepted the MariaDB GPG key, you will be prompted to do so. See "Installing MariaDB with yum" for detailed information.

图 7-1 MariaDB 安装源配置

当网站自动为用户生成Yum源文件后,用户只需将图7-1中虚线方框内的Yum源配置

语句拷贝到 CentOS7 系统的 `/etc/yum.repos.d` 目录下任一个 `.repo` 的 Yum 源文件中，或者在该目录下重新建立一个名为 `MariaDB.repo` 的 Yum 源文件，并将网站生成的 Yum 源配置语句存入其中即可。图 7-1 中生成的 Yum 源配置语句如下：

```
[MariaDB]
name = MariaDB
baseurl = http://yum.MariaDB.org/10.1/centos7-amd64
gpgkey=https://yum.MariaDB.org/RPM-GPG-KEY-MariaDB
gpgcheck=1
```

将网站自动生成的 Yum 源配置语句拷贝到自己的系统后，Yum 源文件 `/etc/yum.repos.d/MariaDB.repo` 的内容如下：

```
[root@controller1-vm ~]#more /etc/yum.repos.d/MariaDB.repo
[MariaDB]
name = MariaDB
baseurl = http://yum.MariaDB.org/10.1/centos7-amd64
gpgkey=https://yum.MariaDB.org/RPM-GPG-KEY-MariaDB
gpgcheck=1
```

Yum 源设置完成之后，使用图 7-1 中提示的 MariaDB 安装命令即可通过 Yum 命令进行在线 MariaDB 软件包的安装。

```
yum install -y MariaDB-server MariaDB-client
```

对于很多用户而言，系统服务器通常位于安全管控区，严禁与 Internet 连接，这种情况下可以通过离线方式安装 MariaDB。采用离线方式安装时，首先需要到 MariaDB 官方 Yum 源网站上下载 RPM 安装包，网址为 `http://yum.MariaDB.org/`。进入该网站之后，即可选择需要安装的 MariaDB 版本和对应的 Linux 发行版本。例如要在 CentOS7 上安装 MariaDB10.1 稳定版本，则需要的 RPM 包位于 `http://yum.MariaDB.org/10.1/centos7-amd64/rpms/` 路径下，如图 7-2 所示。

Index of /10.1/centos7-amd64/rpms/			
Name	Last Modified	Size	Type
Parent Directory/		-	Directory
MariaDB-10.1.14-centos7-x86_64-client.rpm	2016-May-09 16:15:48	38.7M	application/x-redhat-package-manager
MariaDB-10.1.14-centos7-x86_64-common.rpm	2016-May-09 16:15:48	43.1K	application/x-redhat-package-manager
MariaDB-10.1.14-centos7-x86_64-compat.rpm	2016-May-09 16:15:48	1.4M	application/x-redhat-package-manager
MariaDB-10.1.14-centos7-x86_64-connect-engine.rpm	2016-May-09 16:15:47	2.0M	application/x-redhat-package-manager
MariaDB-10.1.14-centos7-x86_64-cracklib-password-check.rpm	2016-May-09 16:15:48	15.7K	application/x-redhat-package-manager
MariaDB-10.1.14-centos7-x86_64-devel.rpm	2016-May-09 16:15:49	6.5M	application/x-redhat-package-manager
MariaDB-10.1.14-centos7-x86_64-gssapi-client.rpm	2016-May-09 16:15:49	18.3K	application/x-redhat-package-manager
MariaDB-10.1.14-centos7-x86_64-oggraph-engine.rpm	2016-May-09 16:15:49	543.8K	application/x-redhat-package-manager
MariaDB-10.1.14-centos7-x86_64-server.rpm	2016-May-09 16:15:52	99.4M	application/x-redhat-package-manager
MariaDB-10.1.14-centos7-x86_64-shared.rpm	2016-May-09 16:15:52	1.2M	application/x-redhat-package-manager
MariaDB-10.1.14-centos7-x86_64-test.rpm	2016-May-09 16:15:54	77.4M	application/x-redhat-package-manager
galera-25.3.15-1.rhel7.el7.centos.x86_64.rpm	2016-Mar-14 20:59:42	7.7M	application/x-redhat-package-manager
jemalloc-3.6.0-1.el7.x86_64.rpm	2015-Feb-14 01:09:56	103.6K	application/x-redhat-package-manager
jemalloc-devel-3.6.0-1.el7.x86_64.rpm	2015-Feb-14 01:09:56	22.3K	application/x-redhat-package-manager

lighttpd/1.4.33

图 7-2 MariaDB RPM 安装位置

从图 7-2 中可以看到, 根据用户的选择, 会出现很多像 server、client、common、shared 这样的 RPM 包, 通常只需要下载 MariaDB-xxx-server.rpm 和 MariaDB-xxx-client.rpm 两个 RPM 包即可。其中, server 后缀的 RPM 包是安装 MariaDB 服务器的, client 后缀的 RPM 包是安装 MariaDB 客户端的。将下载后的 RPM 包上传到 CentOS7 服务器后, 通过如下命令即可安装 MariaDB 的 RPM 包:

```
rpm -ivh MariaDB-10.1.14-centos7-x86_64-server.rpm
rpm -ivh MariaDB-10.1.14-centos7-x86_64-client.rpm
```

这里使用的安装命令是 rpm, 其使用到的安装参数解释如下:

- ❑ -i 参数, 表示安装后面的一个或多个 RPM 软件包。
- ❑ -v 参数, 表示安装过程中显示详细的信息。
- ❑ -h 参数, 表示使用 “#” 符号来显示安装进度。

理想情况下, 最好使用 yum 命令在线安装, 这样 Yum 会自动解析安装包依赖关系并进行依赖包的自动下载安装, 如果是通过 rpm 命令进行安装, 则需要自己准备很多 RPM 依赖包, 建议在使用 rpm 命令安装之前, 将操作系统 ISO 镜像设置为本地 Yum 源, 这样可以避免手动再去下载很多依赖包。

这里使用的 Linux 发行版本是 CentOS7.1-1503 版本, CentOS7.1 中已经集成了 MariaDB5.5 的 RPM 安装包, 由于系统最初采用的是最小安装, 即系统安装时候并没有安装 MariaDB, 为了进行离线安装, 首先需要将 CentOS7.1 的 ISO 安装镜像制作成本地 Yum 源, 将 ISO 解压到 /data/ISO 目录后, 本地 ISO 的 Yum 源配置如下:

```
[root@controller3 ~]# more /etc/yum.repos.d/centos7-iso.repo
[centos7-iso]
name=centos7-iso
baseurl=file:///data/ISO
gpgcheck=0
enabled=1
```

本地 Yum 源配置完成后, 可以通过 yum list 命令检查 Yum 仓库配置是否正确, 并检查 Yum 是否能够从 Yum 仓库中找到 MariaDB 安装包。

```
[root@controller3 ~]# yum list |grep MariaDB*
MariaDB-libs.x86_64           1:5.5.41-2.el7_0      @anaconda
MariaDB.x86_64               1:5.5.41-2.el7_0      centos7-iso
MariaDB-bench.x86_64         1:5.5.41-2.el7_0      centos7-iso
MariaDB-devel.x86_64         1:5.5.41-2.el7_0      centos7-iso
MariaDB-server.x86_64        1:5.5.41-2.el7_0      centos7-iso
MariaDB-test.x86_64          1:5.5.41-2.el7_0      centos7-iso
```

现在即可通过 Yum 进行离线安装 MariaDB, 在 CentOS7 中, MariaDB-client 被打包成了 MariaDB, 因此安装 MariaDB 服务器和 MariaDB 客户端命令的过程如下:

```
[root@controller3 ~]# yum install MariaDB-server MariaDB
```

```

.....
--> Running transaction check
---> Package MariaDB.x86_64 1:5.5.41-2.el7_0 will be installed
---> Package MariaDB-server.x86_64 1:5.5.41-2.el7_0 will be installed
.....
Installed:
MariaDB.x86_64 1:5.5.41-2.el7_0
MariaDB-server.x86_64 1:5.5.41-2.el7_0
Dependency Installed:
perl-Compress-Raw-Zlib.x86_64 1:2.061-4.el7
perl-Compress-Raw-Bzip2.x86_64 0:2.061-3.el7
Complete!

```

可以看到在安装 MariaDB 的同时，还会安装很多 perl 相关的依赖包。MariaDB-server 和 MariaDB 的 RPM 包安装完成后，会在系统的 /etc、/var 和 /usr 目录下新生成很多目录和文件。

```

//客户端RPM包安装后生成的文件
[root@controller3 ~]# rpm -ql MariaDB
// MariaDB客户端配置文件
/etc/my.cnf.d/client.cnf
// /usr/bin目录中的均为可执行文件
/usr/bin/aria_chk
.....
// /usr/share目录中的均为帮助文档
/usr/share/doc/MariaDB-5.5.41
.....
// Server端RPM包安装后生成的文件
[root@controller3 ~]# rpm -ql MariaDB-server
/etc/logrotate.d/MariaDB
//server端配置文件
/etc/my.cnf.d/server.cnf
// /usr/bin目录中的均为MariaDB的可执行命令文件
/usr/bin/innochecksum
.....
//以.so结尾的均为库文件
/usr/lib64/MySQL/plugin/adtd_null.so
.....
// /usr/share目录中的均为MariaDB的帮助文档
/usr/share/man/man1/innochecksum.1.gz
.....
//以.cnf为后缀的是MariaDB自带的配置文件样例
/usr/share/MySQL/my-huge.cnf
.....
//以.sql为后缀的文件是MariaDB自带的测试数据库
/usr/share/MySQL/MySQL_performance_tables.sql
.....
//默认的数据存放目录
/var/lib/MySQL
//默认的日志存放目录

```

```

/var/log/MariaDB
/var/log/MariaDB/MariaDB.log
//进程文件存放路径
/var/run/MariaDB

```

MariaDB 安装完成后，各目录下存放的文件归类如下：

- ❑ /usr/share/MySQL/ 目录下是安装文件和配置文件。
- ❑ /etc/my.cnf.d 目录下是 MariaDB 配置文件。
- ❑ /var/lib/MySQL/ 目录下是 MariaDB 数据库、错误日志和 socket 文件。
- ❑ /usr/share/doc/ 和 /usr/share/man/ 目录下是帮助文档。
- ❑ /usr/bin/ 目录下是 MariaDB 的各种可执行命令。
- ❑ /usr/lib64/MySQL 目录下是 MariaDB 服务运行所需的库文件。

MariaDB 安装完成后，通常无须做任何配置变更即可正常启动并提供数据库服务功能，通常在启动 MariaDB 服务后将其设置为开机自启动：

```

//启动MariaDB
[root@controller3 ~]#systemctl startMariaDB.service
//设置为开机自启动
[root@controller3 ~]#systemctl enableMariaDB.service

```

MariaDB 启动之后，通过 MySQL 命令即可以 root 身份访问 MariaDB 数据库，此时运行 root 以无密码方式访问。进入 MariaDB 数据库后，可以通过 show databases 命令来查看在 MariaDB 中有哪些默认数据库，通过 show engines 命令查看当前版本的 MariaDB 支持哪些数据库引擎。

```

[root@controller3 ~]# MySQL -u root
MariaDB [(none)]> show databases;

```

```

+-----+
| Database          |
+-----+
| information_schema |
| MySQL             |
| performance_schema |
| test              |
+-----+

```

4 rows in set (0.00 sec)

```

MariaDB [(none)]> show engines;

```

```

+-----+-----+-----+
| Engine          | Support | Comment
| Transactions| XA      | Savepoints |
+-----+-----+-----+
| InnoDB          | DEFAULT | Percona-XtraDB, Supports transactions, row-level
locking
| YES             | YES     | YES         |
| CSV             | YES     | CSV storage engine

```

```

| NO          | NO | NO          |
| MRG_MYISAM  |    | YES         | Collection of identical MyISAM tables
| NO          | NO | NO          |
| BLACKHOLE   |    | YES         | /dev/null storage engine (anything you write to it
disappears)
| NO          | NO | NO          |
| MEMORY      |    | YES         | Hash based, stored in memory, useful for tempor-
ary tables
| NO          | NO | NO          |
| PERFORMANCE_SCHEMA | YES | Performance Schema
| NO          | NO | NO          |
| ARCHIVE     |    | YES         | Archive storage engine
| NO          | NO | NO          |
| MyISAM      |    | YES         | MyISAM storage engine
| NO          | NO | NO          |
| FEDERATED   |    | YES         | FederatedX pluggable storage engine
| YES         | NO | YES         |
| Aria        |    | YES         | Crash-safe tables with MyISAM heritage
| NO          | NO | NO          |
+-----+-----+-----+
+-----+-----+-----+
10 rows in set (0.01 sec)

```

可以看到 MariaDB 有 4 个默认数据库，其中 MySQL 是比较重要的数据库，test 只是个测试数据库，可以删除。通过 show engines 命令可以看到当前版本 MariaDB 支持 10 种存储引擎，默认使用的存储引擎为 InnoDB。InnoDB 是一种支持事务处理（Transaction）、支持分布式交易处理的范式（XA）、支持保存点（Savepoint）以便事务回滚的存储引擎。通常在安装完成 MariaDB 后，为了安全起见，必须为数据库的 root 用户设置密码，并禁止 root 进行远程登录。设置 root 密码可以通过 MySQLadmin 完成。

```
[root@controller3 ~]# MySQLadmin -u root password root
```

另外，也可以通过 /usr/bin/MySQL_secure_installation 命令进行 root 密码设置和更改，通过此命令还可以删除 MariaDB 中的测试数据库和匿名用户，以及禁止 root 远程登录的操作，在数据库安装完成之后，建议使用 MySQL_secure_installation 对数据库进行初始化操作。

```

[root@controller3 ~]# MySQL_secure_installation
.....
//输入root密码，没有则按回车键
Enter current password for root (enter for none):
OK, successfully used password, moving on...
.....
// 更改root密码
Change the root password? [Y/n] y
New password:
Re-enter new password:
Password updated successfully!

```



```

Reloading privilege tables...
... Success!
.....

//删除匿名用户
Remove anonymous users? [Y/n] y
... Success!
.....

// 禁止root用户远程登录
Disallow root login remotely? [Y/n] y
... Success!
.....

// 删除test数据库
Remove test database and access to it? [Y/n] y
- Dropping test database...
... Success!
- Removing privileges on test database...
... Success!
.....

//重新加载权限表
Reload privilege tables now? [Y/n] y
... Success!

Cleaning up...
All done! If you've completed all of the above steps, yourMariaDB
installation should now be secure.
Thanks for using MariaDB!

```

为 root 用户设置了密码后，将不再允许未经密码授权的 root 用户登录，而必须通过 -p 参数输入密码才可以访问数据库。

```

[root@controller3 ~]# MySQL -u root
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO)
[root@controller3 ~]# MySQL -u root -p
MariaDB [(none)]> show databases;
+-----+
| Database |
+-----+
| information_schema |
| MySQL |
| performance_schema |
+-----+
3 rows in set (0.00 sec)
MariaDB [(none)]>

```

为 root 用户设置了密码之后，必须提供 root 密码才能访问数据库，而且通过 MySQL_secure_installation 命令删除 test 数据库后，现在只剩下 MySQL、performance_schema、information_schema 三个默认数据库。在上面的 MariaDB 启动过程中，我们并没有对 MariaDB 进行任何自定义的配置，而是采用了默认配置。MariaDB 对配置文件的读取顺序为 /etc/MySQL/my.cnf，/etc/my.cnf，~/my.cnf。在 CentOS71 环境中，配置文件主要是 /

etc/my.cnf, 当然用户也可以在自己的主目录下创建 my.cnf 文件, 同时屏蔽其他目录下的 my.cnf 文件。因此在 CentOS71 环境中, 如果用户需要自定义 MariaDB 的配置, 可以修改配置文件 /etc/my.cnf, 此文件的内容大致如下:

```
[root@controller3 etc]# more/etc/my.cnf
[MySQLd]
datadir=/var/lib/MySQL
socket=/var/lib/MySQL/MySQL.sock
# Disabling symbolic-links is recommended to prevent assorted security risks
symbolic-links=0
# Settings user and group are ignored when systemd is used.
# If you need to run MySQLd under a different user or group,
# customize your systemd unit file for MariaDB according to the
# instructions in http://fedoraproject.org/wiki/Systemd

[MySQLd_safe]
log-error=/var/log/MariaDB/MariaDB.log
pid-file=/var/run/MariaDB/MariaDB.pid
# include all files from the config directory
!includedir /etc/my.cnf.d
```

/etc/my.cnf 中最重要的两个参数是 datadir 和 socket, 可以看到该文件的最后一行是个文件包含命令, 被包含的目录是 /etc/my.cnf.d, 即 MariaDB 除了读取 my.cnf 文件配置参数外, 还会读取该目录下以 .cnf 结尾的配置文件。

```
[root@controller3 my.cnf.d]# ls -l /etc/my.cnf.d
total 12
-rw-r--r-- 1 root root 295 Dec 19 2014 client.cnf
-rw-r--r-- 1 root root 232 Dec 19 2014 MySQL-clients.cnf
-rw-r--r-- 1 root root 744 Dec 19 2014 server.cnf
```

在 /etc/my.cnf.d 目录中, client.cnf 主要用于配置 MariaDB 客户端, 只有客户端才会读取该文件。MySQL-clients.cnf 主要用于配置 MariaDB 的命令行工具, 可以将常用的命令行参数配置到该文件中对应的命令行工具段, 在使用该命令行工具时会自动读取该配置参数, 该文件默认为空。server.cnf 主要用于配置 MariaDB 的 Server 端, 客户端不会读取该配置文件。在以上几个配置文件中, 对 MariaDB 使用影响最大的是 server.cnf 文件, 该文件可以参考 /usr/share/MySQL 下的 .cnf 配置样例文件进行配置。

```
[root@controller3 MySQL]# pwd
/usr/share/MySQL
[root@controller3 MySQL]# ls -l |grep cnf
-rw-r--r-- 1 root root 4920 Feb 5 2015 my-huge.cnf
-rw-r--r-- 1 root root 20438 Feb 5 2015 my-innodb-heavy-4G.cnf
-rw-r--r-- 1 root root 4907 Feb 5 2015 my-large.cnf
-rw-r--r-- 1 root root 4920 Feb 5 2015 my-medium.cnf
-rw-r--r-- 1 root root 2846 Feb 5 2015 my-small.cnf
-rw-r--r-- 1 root root 656 Nov 17 2014 README.MySQL.cnf
```

/usr/share/MySQL 目录下提供了 small、medium、large、huge 四种类型的配置模板，分别用于配置小型、中型、大型和巨型 MariaDB 数据库。由于 MariaDB 的默认配置文件为 /etc/my.cnf，而该配置文件中最后有一句“!includedir /etc/my.cnf.d”，该语句的作用就是将 /etc/my.cnf.d 目录下文件的内容追加到当前文件即 /etc/my.cnf 中，因此，假如需要部署一套大型数据库环境，则只需将 my-huge.cnf 拷贝到 /etc/my.cnf.d/ 目录下，如果部署小型 MariaDB 数据库，则只需将 my-small.cnf 拷贝到该目录下，如果仅使用 InnoDB 引擎并且 RAM 为 4GB，同时连接较少但是存在大量查询，则将 my-innodb-heavy-4G.cnf 拷贝到该目录。当 MariaDB 启动读取配置文件的时候，/etc/my.cnf.d 目录下的文件内容都会追加到默认配置文件 /etc/my.cnf 中而被读取。

7.1.4 MariaDB 高可用方案

数据库作为一个系统最为核心的组成部分，实现零宕机或者 7×24 小时不间断持续服务几乎是每一款数据库产品的必备功能。MariaDB 也不例外，作为全兼容 MySQL 的数据库，MariaDB 在高可用上不仅继承了 MySQL 高可用（MHA，MySQL High Availability）功能，如 MySQL 的数据复制（Replication）功能，同时还新增了 MaxScale 等 MariaDB 团队为其开发的高可用组件。在 MariaDB 和 MySQL 社区，最为活跃的话题之一便是数据库的高可用性讨论。针对用户不同的使用场景，可以有不同的数据库高可用实现方式，本节主要介绍基于 MariaDB 的数据库高可用方案。通常，对于数据库的高可用而言，有三个最基本和核心的功能必须被实现，即数据冗余、监控管理和故障切换机制。

- ❑ 数据冗余：数据冗余意味着数据需要在多台服务器之间进行同步，并且这些服务器在必要的时候可以访问相同的数据，从而保证在数据库主服务器故障的情况下，其他备节点可以迅速接管实时同步的数据并继续提供数据库访问功能。
- ❑ 监控管理：在高可用集群中，监控和管理模块负责对整个集群服务运行状态进行监控，并在发现节点或服务故障后进行相应的资源调度和故障切换等管理功能。因此，对于数据库高可用集群而言，必须提供一个稳定的监控和管理模块来跟踪和控制服务，以便在发生节点或服务故障的情况下能够对集群服务进行相应的故障转移。
- ❑ 故障切换：对任何高可用集群而言，故障切换（Failover）机制是最重要的高可用实现保证，在节点或服务出现故障的情况下，Failover 机制能够将故障节点或服务切换至正常节点运行，并在故障节点恢复后，将服务切回该节点或者保持原状不变。在服务的故障转移过程中，Failover 机制应该将该服务的访问路由至新的服务节点，以保证客户端对整个故障切换过程的透明性，即客户端对集群内部的故障处理是无感知的。

每一个数据库软件都有针对自身的高可用设计，如 IBM DB2 数据库的 PureScale，Oracle 数据库的 RAC 等功能都是利用集群方式实现的数据库高可用。而在 MariaDB 或

MySQL 的高可用部署中, 也存在几种主流的数据冗余高可用方案, 分别是共享存储方案、MySQL Replication 方案和 Galera 集群方案。

❑ 共享存储或磁盘复制: 共享存储方案是很普遍也非常经典的数据冗余方案, 但是部署实施过程相对繁琐复杂, 并且需要额外的存储支持故成本相对过高。这种方案最常见的实现方式便是 SAN 存储或者基于 Linux 的磁盘复制技术, 如 DRDB 技术。MariaDB 也支持这种数据冗余方式, 但在大多数的数据库环境中, 并不推荐使用共享存储或者磁盘复制方式来实现数据冗余, 因为其实实施相对复杂并且仅是数据级别的冗余, 故障时数据库的恢复过程也相对繁琐。

❑ MySQL Replication: 数据复制 Replication 最初是 MySQL 自带的基于软件的数据复制技术, Replication 方案的优点在于, 基于软件实现所以实施过程相对简单, 并且成本开销较低。最初在 MySQL 中开发 Replication 功能的初衷是实现数据库集群的 Scale-out 扩展, 因此, MySQL Replication 技术在集群使用中并不能真正实现可靠的高可用性。MariaDB 支持用户使用 MySQL Replication 技术实现数据冗余, 通常这也是非常普遍的使用方式, 但是 Replication 技术在 MySQL 集群高可用中使用的相对较少, 主要用于 MySQL 数据库集群的 Scale-out 扩展场景中。当 MySQL Replication 用于集群高可用部署时, 其主要与 MySQL 的高可用技术 MHA 结合使用。

❑ Galera 集群: Galera 是一种用于 MariaDB/MySQL 中的数据库同步复制软件, 并且是完全独立于经典的 MySQL Replication 集群技术。在使用 Galera 的集群中, 允许数据库节点实时添加和移除, 并且集群中每个数据库节点的数据随时保持一致性, 而各个节点的状态均被作为全局状态进行维护。Galera 是一种多主 (Multi-Master) 复制技术, 多主复制的部署使得 Failover 的过程更为简单和高效。MariaDB 将 Galera 当作一个标准组件包含在 MariaDB Galera Cluster 中对外发布, 同时作为独立产品的 MDBE (MariaDB Enterprise) 和 MaxScale 也都支持 Galera 集群功能。

在高可用数据库集群中, 除了数据冗余, 还需要在高可用集群中实现集群节点和服务的监控管理与故障切换机制。在服务监控管理领域存在多种不同的技术实现, 包括基于硬件的负载均衡技术、基于第三方软件的访问代理实现和针对特定应用软件开发集群管理软件等。在 MariaDB/MySQL 数据库高可用部署中, 主流的集群监控管理和故障 Failover 技术主要包括硬件负载均衡技术、基于第三方软件的负载均衡技术、MHA 技术、基于驱动和应用程序的故障切换技术以及 MaxScale 等技术。

❑ 硬件负载均衡: 在部分高可用部署中会使用到硬件负载均衡技术, 如 F5 公司的 BigIP 技术。这种部署方式适于基于磁盘的 Failover 解决方案和使用 Galera 的场景, 而且实际使用效果也比较符合生产要求。MariaDB 支持硬件负载均衡器部署方案, 但是在该方案下, 负载均衡器的配置和其他一些相关的配置需要负载均衡器厂商提供。

- ❑ 集成第三方软件：在 Linux 系统中，HAProxy 是一款基于软件协议实现的开源负载均衡器和 Failover 软件，在集群系统中有着非常广泛的使用。而在 Windows 系统中，Windows Server Failover Cluster 也具有类似的集群管理与故障切换功能。MariaDB 支持使用这类基于第三方软件的高可用解决方案，但是和使用硬件负载均衡器一样，MariaDB 也并非完全兼容这些三方组件。在使用 HAProxy 的场景中，MariaDB 在某些方面能够兼容和支持 HAProxy 的设置与使用，或者说 MariaDB 是亲和 HAProxy 的。
- ❑ MHA：MHA (MySQL High Availability) 的主要功能是结合 MySQL Replication 技术以实现 MariaDB 的高可用集群方案，由于 MySQL Replication 的异步复制和配置方式，使得要在基于 MySQL Replication 的集群中实现故障切换比较困难，因此单独的 MySQL Replication 高可用方案存在诸多不足和缺陷。而 MHA 便是专门解决和弥补 MySQL Replication 不足与缺点的技术，在某些情况下，MHA 也是一种简单有效的高可用解决方案。而 MariaDB 作为 MHA 技术的社区维护者，完全支持 MHA 功能。
- ❑ 基于驱动和应用的 Failover：在某些高可用环境中，尤其是较为简单的集群配置环境中，数据库驱动甚至应用程序都支持 Failover，高可用解决方案的监控和管理则通过其他的一些方法来实现。例如在 MariaDB 中，支持使用标准 MySQL 驱动（也称为 Connectors）进行 Failover。
- ❑ MaxScale：MaxScale 是一种应用较为广泛且部署于 MariaDB 服务器端的技术，主要功能是监控服务器状态并实现负载均衡，同时也提供 Failover 功能和对通信接口进行监控的功能。MaxScale 还支持 API 调用，这意味着 MaxScale 可以整合到其他基础架构组件和技术栈中。MariaDB 社区是 MaxScale 的开发者，同时完全支持 MaxScale 的使用。

MariaDB 已形成了自己比较完善的软件生态，从 MariaDB 数据库社区版服务器端、客户端和集群高可用软件，到企业版的 MariaDB 数据库软件和集群高可用管理软件，具有比较完整的软件产品发布。从目前 MariaDB 的官方网站来看，MariaDB 社区主要提供的软件产品包括社区版的数据库软件 MariaDB Database Server 和集群管理软件 MariaDB Galera Cluster，以及企业版的数据库软件 MariaDB Enterprise 和集群管理软件 MaxScale。其中，MariaDB Server 是核心数据库服务器；MariaDB Galera Cluster 是目前使用最为广泛的 MariaDB 数据库高可用软件；MaxScale 是位于应用和数据库之间的通用核心组件，主要用于分片、Failover、负载均衡、监控和审计等功能；MariaDB Enterprise 是基于 MariaDB Galera Cluster 开发的企业版数据库软件，除了提供高可用解决方案之外，还提供监控管理和 GUI 等功能。针对多数用户环境，选择的产品组合主要是 MariaDB Server 和 MariaDB Galera Cluster，以提供基本的数据库功能和高可用功能，当然，也可以选择 MaxScale 和 MariaDB Enterprise 软件来实现更精细和多功能的部署与管理。

7.1.5 MariaDB Galera Cluster 概述

MariaDB Galera Cluster 是 MariaDB 的一个多主 (Multi-Master) 同步集群软件, 使用 Galera Cluster 可以让 MariaDB 集群的所有节点保持数据同步, 并且每个节点都可以对外提供服务。相对于主从 (Master-Slave) 异步复制, Galera 为 MariaDB 提供了同步数据复制功能, 因此其可以保证集群中的数据冗余和节点之间的数据一致性, MariaDB Galera Cluster 当前仅支持 InnoDB 存储引擎 (MyISAM 引擎属于实验性的扩展支持, 且只有在 Linux 系统下可以使用该集群软件)。Galera Cluster 具有如下特性:

- ❑ 同步复制 (Synchronous Replication), 各节点间无延迟且节点宕机不会导致数据丢失。
- ❑ Active-Active 的 Multi-Master 多主逻辑拓扑, 节点之间互为热备, 在 Failover 过程中无任何中断时间。
- ❑ 真正的多主架构, 集群中可以并存多个 Master 节点, 可对集群中任一节点进行数据读写。
- ❑ 自动成员控制, 故障节点自动从集群中移除。
- ❑ 自动节点加入, 无须手工备份当前数据库并拷贝至新节点。
- ❑ 真正并行的复制, 复制基于 Row 级别。
- ❑ 直接客户端连接, 原生的 MySQL 接口。
- ❑ 每个节点都包含完整的数据副本, 无须读写分离。
- ❑ 多台数据库中数据同步由 wsrep 接口实现。

由于 Galera 的上述特性, 使得基于 Galera 集群的 MariaDB 数据库具有很多优势, 例如多主同步机制使得节点故障时客户端访问无须切换, 即无须 Failover 的过程, 因此不存在故障切换时的访问中断情况。由于可以同时读写多个节点, 这相当于将数据库的访问带宽进行了 Scale-out 扩容, 从而允许数据库支撑更多的连接访问。当然, 目前 Galera 也存在一些局限性 (在使用 Galera 之前, 需要考虑清楚自己的数据库环境是否会有因使用 Galera 而出现异常的情况) 其局限性表现在如下几方面:

- ❑ 目前的复制仅支持 InnoDB 存储引擎, 任何写入其他引擎的表, 包括写入 MySQL.* 表的数据将不会被复制, 但是 Galera 会复制 DDL 语句的执行结果, 因此创建的用户将会被复制, 但是像 “INSERT INTO MySQL.user...” 这样的执行结果将不会被复制, 因为 DELETE 操作不支持没有主键的表 (没有主键的表在不同的节点顺序将不同, 如果执行 “SELECT...LIMIT...” 将出现不同的结果集)。
- ❑ 在多主环境下不支持 LOCK/UNLOCK TABLES, 也不支持锁函数 GET_LOCK 和 RELEASE_LOCK, 此外, 查询日志不能保存在表中, 如果开启查询日志, 只能保存到文件中。
- ❑ 允许的最大事务大小由 wsrep_max_ws_rows 和 wsrep_max_ws_size 定义, 任何大型数据操作将被拒绝, 如大型的 LOAD DATA 操作。
- ❑ 由于集群是乐观锁并发控制, 事务提交可能在该阶段中止。如果有两个事务向集群

中不同的节点相同表的同一行写入并提交，失败的节点将会被中止。

- ❑ 不支持集群 XA 事务，因为在提交上可能回滚。
- ❑ 整个集群的写入吞吐量是由最弱的节点限制，如果有一个节点变得缓慢，那么整个集群将是缓慢的，为了稳定的高性能要求，所有的节点应使用统一的硬件。
- ❑ 建议集群最少由 3 个节点构成。
- ❑ 如果 DDL 语句有问题则整个集群都会受到破坏。

Galera Cluster 是一种真正的多主同步复制集群，因此客户端可以对集群中的任一个节点进行读写操作，由于数据在各个节点之间的同步一致性，客户端在任一个节点上读取到的数据都是相同的，同时客户端写入任一个节点的数据都会被自动同步到其他节点，Galera Cluster 的工作原理架构如图 7-3 所示。

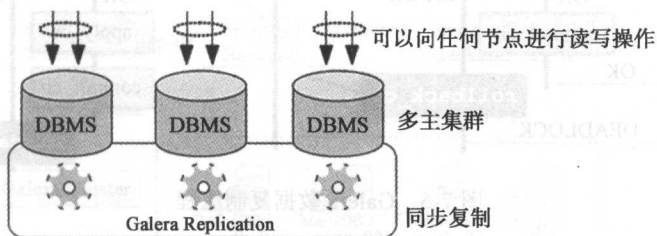


图 7-3 MariaDB Galera 原理架构

Galera 集群的复制功能基于 Galera Library 实现，为了让 MariaDB/MySQL 与 Galera-Library 通信，特别针对 MariaDB/MySQL 开发了 WSREP API，Galera Cluster 中的功能模块结构如图 7-4 所示。

Galera 集群软件能够保证集群节点之间数据库的同步，并保持数据一致性，其中的关键环节便是图 7-4 中的认证（Certification）功能模块。Galera 集群在数据复制之前的认证功能确保了所有提交到数据库节点的事务均是完整正确的。图 7-5 和图 7-6 从不同的角度描述了 Galera 集群中不同节点之间的数据认证复制过程和认证复制工作原理。

在图 7-5 和图 7-6 中，当客户端发出一个 commit 指令时，在事务被提交之前，所有对数据库的更改都会被 write-set 收集起来，并将 write-set 收集的内容发送给其他节点，图 7-6 中的 S0、S1 和 S2 分别表示三个不同的数据服务器节点。write-set 将在每个节点进行认证测试，认证结果决定了节点是否应用 write-set 中的事务更改本节点数据库中的数据。如果认证测试失败，节点将丢弃 write-set 中的事务；如果认证测试成功，则 write-set 中的事务被提交。认证测试在 Galera 集群中的实现取决于全局事务顺序，每个事务在复制期间都会被指派一个全局事务顺序（Global

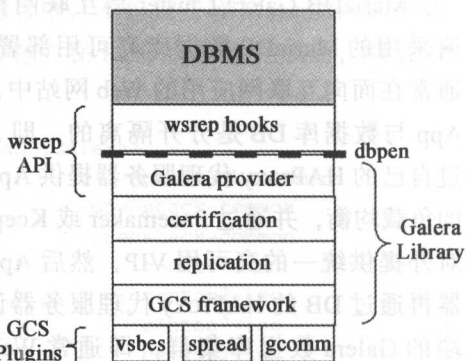


图 7-4 Galera 集群功能层次模块

trx ID，如图 7-5 所示)。当一个事务到达提交点时，Galera 根据全局事务顺序 ID 来进行事务相关的主键冲突检测，如果发现冲突认证测试就会失败。图 7-7 描述了 MariaDB 数据库管理器调用 wsrep API 进行数据复制的过程。

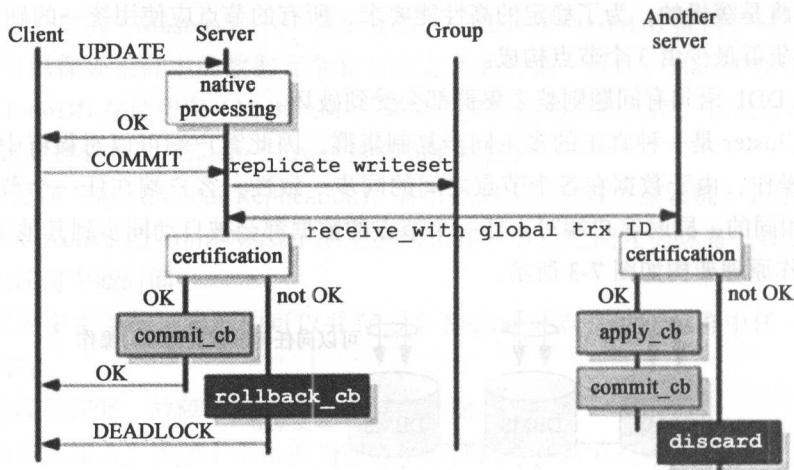


图 7-5 Galera 数据复制流程

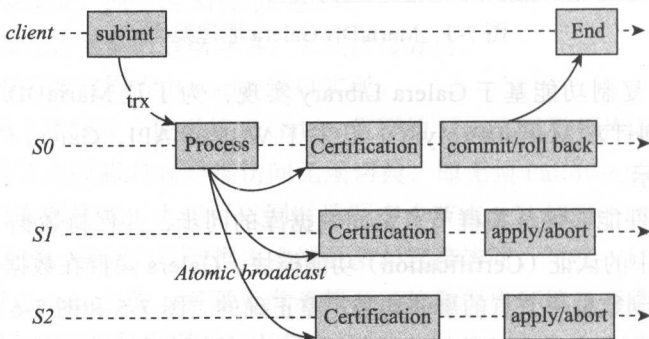


图 7-6 Galera 数据认证工作原理

MariaDB Galera Cluster 是互联网行业普遍采用的 MariaDB 数据库高可用部署方案。通常在面向互联网应用的 Web 网站中，应用 App 与数据库 DB 是分开隔离的，即 App 通过自己的 HAProxy 代理服务器提供 App 访问的负载均衡，并通过 Pacemaker 或 Keepalived 对外提供统一的高可用 VIP，然后 App 服务器再通过 DB 的 HAProxy 代理服务器访问后端的 Galera 数据库集群，即通常 Web 网站是两层或三层甚至更多层次的体系架构，而

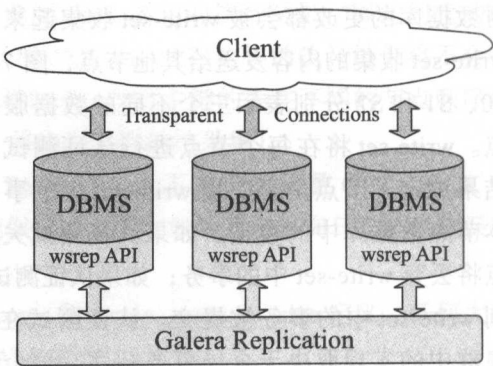


图 7-7 Galera 集群基于 wsrep API 的 Replication

MariaDB Galera 集群数据库总是位于架构的最底层。图 7-8 显示了 MariaDB Galera 集群在这种典型 Web 应用架构中的应用。

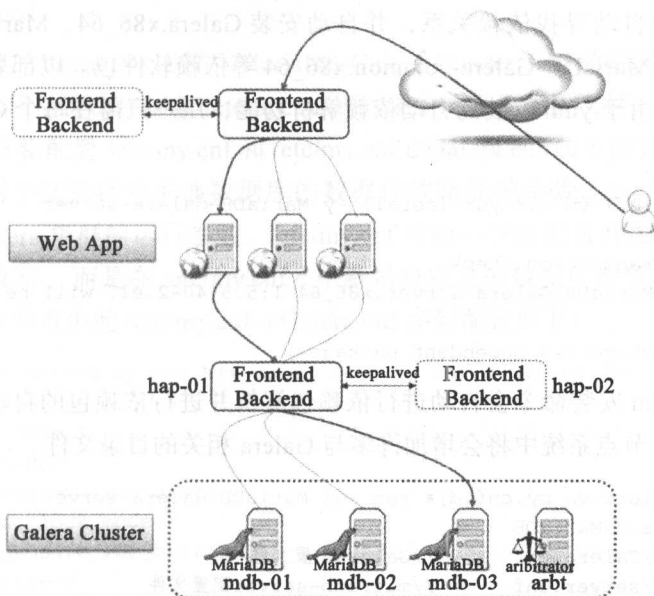


图 7-8 MariaDB 在典型 Web 网站架构中的应用

7.1.6 MariaDB Galera Cluster 配置

在 MariaDB 的高可用集群部署中，主流的实现方案还是 Galera Cluster 与 HAProxy 和 Pacemaker 的集成方案。在本书中采用的也是基于 HAProxy 和 Pacemaker 的 MariaDB Galera 集群方案。其中，Galera Cluster 负责数据库节点之间的数据同步和一致性，Pacemaker 负责 MariaDB 资源服务的启停和运行监控等功能，HAProxy 主要负责数据库对外服务的负载均衡。MariaDB Galera Cluster 部署的第一步就是软件安装，在 RHEL 和 CentOS 版本的 Linux 系统中，建议通过 Redhat 的 RDO 源进行 MariaDB Galera Cluster 的部署。与部署单机版的 MariaDB 数据库不同，在 Galera 集群部署中，不再是安装 MariaDB-server 软件包，而是安装 MariaDB-Galera-server 和 Galera 软件包。在 RDO 源配置完成后，将会看到 yum 仓库中新增了上述两个软件包。

```
[root@controller1-vm ~]# yum list |egrep "MariaDB|Galera"
```

Galera.x86_64	25.3.5-6.el7	openstack-common
MariaDB-Galera-common.x86_64	1:5.5.40-2.el7	openstack-common
MariaDB-Galera-server.x86_64	1:5.5.40-2.el7	openstack-common
MariaDB.x86_64	1:5.5.41-2.el7_0	centos7-iso
MariaDB-bench.x86_64	1:5.5.41-2.el7_0	centos7-iso
MariaDB-devel.x86_64	1:5.5.41-2.el7_0	centos7-iso
MariaDB-libs.x86_64	1:5.5.41-2.el7_0	centos7-iso
MariaDB-server.x86_64	1:5.5.41-2.el7_0	centos7-iso

```
MariaDB-test.x86_64 1:5.5.41-2.el7_0 centos7-iso
```

在上述软件包中, 用户只需使用 `yum` 命令安装 `MariaDB-Galera-server.x86_64` 软件包即可, `yum` 命令会自动寻找依赖关系, 并自动安装 `Galera.x86_64`、`MariaDB-libs.x86_64`、`MariaDB.x86_64`、`MariaDB-Galera-common.x86_64` 等依赖软件包。以部署三节点 MariaDB Galera 集群为例, 由于 `yum` 安装的自动依赖解析功能, 用户只需在每个 Galera 节点运行如下安装命令即可:

```
[root@controller1-vm ~]# yum install -y MariaDB-Galera-server
.....
--> Running transaction check
--> Package MariaDB-Galera-server.x86_64 1:5.5.40-2.el7 will be installed
.....
Install 1 Package (+5 Dependent packages)
```

可以看到, `yum` 安装命令会自动进行依赖包解析并进行依赖包的自动安装, 安装完成之后, 每个 Galera 节点系统中将会增加许多与 Galera 相关的目录文件。

```
[root@controller1-vm my.cnf.d]# rpm -ql MariaDB-Galera-server
/etc/logrotate.d/MariaDB
/etc/my.cnf.d/Galera.cnf          //Galera配置文件
/etc/my.cnf.d/server.cnf         //mairadb-server配置文件
/etc/sysconfig/Clustercheck      //Cluster检查配置文件
/usr/bin/Clustercheck            //数据库相关的可执行命令
.....
/usr/bin/wsrep_sst_common        //write-set相关的可执行命令
/usr/bin/wsrep_sst_MySQLdump
/usr/bin/wsrep_sst_rsync
.....
/usr/lib64/MySQL/plugin/adt_null.so      //库文件
/usr/lib64/MySQL/plugin/auth_0x0100.so
.....
//下述为MariaDB-Galera-server配置模板
/usr/share/doc/MariaDB-Galera-server-5.5.40/my-huge.cnf
/usr/share/doc/MariaDB-Galera-server-5.5.40/my-innodb-heavy-4G.cnf
.....
/usr/share/man/man1/innochecksum.1.gz    //帮助文档
/usr/share/man/man1/msql2MySQL.1.gz
.....
/usr/share/MariaDB-Galera/my-huge.cnf    //MariaDB-Galera配置模板
/usr/share/MariaDB-Galera/my-innodb-heavy-4G.cnf
.....
/usr/share/MariaDB-Galera/MySQL_performance_tables.sql //测试数据库
/usr/share/MariaDB-Galera/MySQL_system_tables.sql
.....
/usr/share/MariaDB-Galera/wsrep.cnf      //wsrep配置模板
/usr/share/MariaDB-Galera/wsrep_notify
/var/lib/MySQL                          //数据库文件目录
/var/log/MariaDB                        //日志文件
```



```

/var/log/MariaDB/MariaDB.log
/var/log/MySQLd.log
/var/run/MariaDB
/var/run/MySQLd

```

与部署单机版的 MariaDB 一样, MariaDB Galera Cluster 在启动时默认读取的配置文件是 /etc/my.cnf, 然后从该文件加载 /etc/my.cnf.d 目录中的其他配置文件。因此, 对于 Galera Cluster, 用户只需要配置 /etc/my.cnf 和 /etc/my.cnf.d/Galera.cnf 两个配置文件即可。其中, /etc/my.cnf 主要用于配置针对本地数据库的数据存放路径等参数, /etc/my.cnf.d/Galera.cnf 主要用于配置 Galera 集群的运行参数。/etc/my.cnf 与前一节的配置并无区别, 通常不建议直接修改该配置文件, 而是在 /etc/my.cnf.d 中新增自定义的数据库配置。Galera Cluster 安装完成后, Galera 节点中的 /etc/my.cnf.d/Galera.cnf 参数配置如下:

```

[root@controller1-vm my.cnf.d]# more /etc/my.cnf.d/Galera.cnf
[MySQLd]
skip-name-resolve=1
binlog_format=ROW
default-storage-engine=innodb
innodb_autoinc_lock_mode=2
innodb_locks_unsafe_for_binlog=1
query_cache_size=0
query_cache_type=0
bind_address=192.168.142.110 //Galera监听地址, 不同节点监听地址不一样
wsrep_provider=/usr/lib64/Galera/libGalera_smm.so
wsrep_Cluster_name="Galera_Cluster"
wsrep_slave_threads=1
wsrep_certify_nonPK=1
wsrep_max_ws_rows=131072
wsrep_max_ws_size=1073741824
wsrep_debug=0
wsrep_convert_LOCK_to_trx=0
wsrep_retry_autocommit=1
wsrep_auto_increment_control=1
wsrep_drupal_282555_workaround=0
wsrep_causal_reads=0
wsrep_notify_cmd=
wsrep_sst_method=rsync

```

对于简单的 Galera 集群而言, 只需修改配置各个节点的 /etc/my.cnf.d/Galera.cnf 文件即可。MariaDB Galera Cluster 的配置完成后, 为了实现集群自动监控管理和负载均衡, 还需要配置 Pacemaker 和 HAProxy。其中, HAProxy 用于提供 Galera 集群的负载均衡功能, Pacemaker 用于监控管理 Galera 集群。假设三个节点的主机名称分别为 controller1-vm、controller2-vm 和 controller3-vm, 对应的 IP 地址段为 192.168.142.110~112, MariaDB 监听默认的 3306 端口, 则 HAProxy 负载均衡器针对 Galera 集群的配置文件如下:

```

[root@controller1-vm ~]# more /etc/Haproxy/Haproxy.cfg

```

```

.....
frontend vip-db                                //负载均衡前端
    bind 192.168.142.201:3306                  //提供客户端访问的虚拟IP
    timeout client 90m
    default_backend db-vms-Galera
.....
backend db-vms-Galera                          //负载均衡后端
    option httpchk
    option tcpka
    stick-table type ip size 1000              //使用stich-table选项
    stick on dst
    timeout server 90m
    server controller1-vm 192.168.142.110:3306 check inter 1s port 9200 backup
        on-marked-down shutdown-sessions      //后端数据库服务器1
    server controller2-vm 192.168.142.111:3306 check inter 1s port 9200 backup
        on-marked-down shutdown-sessions      //后端数据库服务器2
    server controller3-vm 192.168.142.112:3306 check inter 1s port 9200 backup
        on-marked-down shutdown-sessions      //后端数据库服务器3
.....

```

HAProxy 配置完成后, 需要在 Pacemaker 中创建两个资源, 一个是 HAProxy 中指定用于对外提供数据库服务的虚拟 IP 地址资源, 另一个便是 Galera 多主状态资源。由于三个节点已经是 Pacemaker 的成员, 因此资源配置只需在任一节点上运行即可:

```

//虚拟IP资源
[root@controller1-vm ~]# pcs resource create vip-db IPAddr2 ip=192.168.142.201
    nic=eth1
//集群资源
[root@controller1-vm ~]# pcs resource create GaleraGalera enable_creation=true
    wsrep_Cluster_address="gcomm://{node_list}" additional_parameters="--open-
    files-limit=16384' meta master-max=3 ordered=true op promote timeout=300s on-
    fail=block --master

```

在配置 Galera 资源时, 特别注意 `wsrep_Cluster_address="gcomm://{node_list}"` 的写法, 在这里的三节点部署环境中, `wsrep_Cluster_address` 应该写成如下形式:

```

wsrep_Cluster_address="gcomm://{controller1-vm, controller2-vm, controller3-vm}"
//节点之间用逗号隔开

```

在基于 HAProxy 和 Pacemaker 的 Galera 集群中, 应用程序对 Galera 数据库集群的访问 IP 由 HAProxy 提供 (如此处的数据库访问地址为 192.168.142.201:3306), 并且 HAProxy 将从该地址监听到的数据转发到后端数据库服务器, 后端三个数据库节点都为 Master 节点 (Multi-Master), 各个节点之间由 Galera 实现数据的同步。在 Pacemaker 集群中, 当 Galera 资源创建完成并成功启动后, 将会看到如下资源状态:

```

Master/Slave Set: Galera-master [Galera]
Masters: [ controller1-vm controller2-vm controller3-vm]
Stopped: [ computer1 computer2 ]

```

从 Pacemaker 的资源状态中可以看到, 集群中的三个节点均为 Master 节点, 即 Multi-Master 多主节点。要查看 Galera Cluster 的运行状态, 可以登录到任一 MariaDB 数据库节点, 通过命令 “show status like 'wsrep_%'” 进行查看, 并且在正常情况下, 三个节点上的集群运行状态应该保持一致:

```
//查看Galera集群运行状态
MariaDB [(none)]> show status like "wsrep_%";
```

Variable_name	Value
wsrep_local_state_uuid	3c02a77e-e2d4-11e5-af4e-623059392f54
wsrep_protocol_version	5
wsrep_last_committed	1408921
wsrep_replicated	2
wsrep_replicated_bytes	396
wsrep_repl_keys	2
wsrep_repl_keys_bytes	62
wsrep_repl_data_bytes	206
wsrep_repl_other_bytes	0
wsrep_received	2206
wsrep_received_bytes	1069372
wsrep_local_commits	0
wsrep_local_cert_failures	0
wsrep_local_replays	0
wsrep_local_send_queue	0
wsrep_local_send_queue_avg	0.000000
wsrep_local_recv_queue	0
wsrep_local_recv_queue_avg	0.116500
wsrep_local_cached_downto	1406734
wsrep_flow_control_paused_ns	0
wsrep_flow_control_paused	0.000000
wsrep_flow_control_sent	0
wsrep_flow_control_recv	0
wsrep_cert_deps_distance	1.036563
wsrep_apply_oooe	0.000000
wsrep_apply_ool	0.000000
wsrep_apply_window	1.000000
wsrep_commit_oooe	0.000000
wsrep_commit_ool	0.000000
wsrep_commit_window	1.000000
wsrep_local_state	4
wsrep_local_state_comment	Synched
wsrep_cert_index_size	56
wsrep_causal_reads	0
wsrep_cert_interval	0.110603
wsrep_incoming_addresses	192.168.142.110:3306,192.168.142.111:3306, 192.168.142.112:3306
wsrep_Cluster_conf_id	11
wsrep_Cluster_size	3

```

| wsrep_Cluster_state_uuid | 3c02a77e-e2d4-11e5-af4e-623059392f54 |
| wsrep_Cluster_status    | Primary |
| wsrep_connected         | ON |
| wsrep_local_bf_aborts   | 0 |
| wsrep_local_index       | 1 |
| wsrep_provider_name     | Galera |
| wsrep_provider_vendor   | Codership Oy <info@codership.com> |
| wsrep_provider_version  | 3.5(rXXXX) |
| wsrep_ready             | ON |
| wsrep_thread_count      | 2 |
+-----+-----+

```

48 rows in set (0.01 sec)

在“show status like “wsrep_%””命令的输出中，如果 wsrep_ready 为 ON，则说明 Galera Cluster 已经正常运行并可接收数据访问请求。wsrep_Cluster_size 代表 Galera Cluster 集群节点的个数，这里为 3 个。wsrep_incoming_addresses 给出了加入 Galera Cluster 集群的节点 IP 地址，这里 wsrep_incoming_addresses 为 192.168.142.111:3306、192.168.142.110:3306 和 192.168.142.112:3306，正好对应三个后端数据库节点的 IP 和端口。除了上述几个 Galera Cluster 变量外，如下的集群变量在故障诊断和集群运行状况分析中也具有重要的参考价值。

- ❑ wsrep_Cluster_state_uuid：Galera 集群状态 ID，集群中所有节点上该值应该是相同的，如果当前节点上该值与其他节点不同，说明当前节点没有加入其他节点所在的 Galera 集群。
- ❑ wsrep_Cluster_conf_id：集群节点关系改变的次数（每次增加 / 删除都会+1），正常情况下所有节点上该值是相同的，如果值不同，说明该节点被集群临时隔离，当节点之间网络连接恢复时，此参数值应该会恢复一致。
- ❑ wsrep_Cluster_size：Galera Cluster 集群节点个数，如果这个值跟预期规划的节点数一致，则表明所有的集群节点都已经加入集群，否则可能有节点已经故障并被移除。
- ❑ wsrep_Cluster_status：Galera Cluster 集群状态，如果不为“Primary”，则说明集群出现了分区（Partition）或是脑裂（Split-brain）的情况。
- ❑ wsrep_ready：集群是否准备完毕的标志，如果该值为 ON，则说明集群可以提供 SQL 访问负载，如果为 Off，则需要检查 wsrep_connected 的值以进一步判断集群状态。
- ❑ wsrep_connected：如果该值为 Off，且 wsrep_ready 的值也为 Off，则说明该节点没有连接到集群。
- ❑ wsrep_local_state_comment：如果 wsrep_connected 为 On，但 wsrep_ready 为 OFF，则该参数提供了集群不正常的具体原因。
- ❑ wsrep_flow_control_paused：该参数表示集群数据复制停止了多长时间，值为 0~1，

越靠近 0 越好，值为 1 表示复制过程完全停止，优化 `wsrep_slave_threads` 变量可以改善集群数据的复制情况。

- ❑ `wsrep_cert_deps_distance`：此参数表示有多少事务可以并行应用处理，此外，`wsrep_slave_threads` 参数设置的值不应该高出该参数值太多。
- ❑ `wsrep_flow_control_sent`：此参数表示当前节点已经停止了多少次数据同步复制，即集群中出现了数据不一致的节点。
- ❑ `wsrep_local_recv_queue_avg`：此参数表示 Slave 事务队列的平均长度，这是 Slave 瓶颈的预兆。如果节点处理事务能力越慢，则 `wsrep_flow_control_sent` 和 `wsrep_local_recv_queue_avg` 两个参数值越高，这两个值较低的话，说明事务处理相对较好。
- ❑ `wsrep_local_send_queue_avg`：此参数代表了网络瓶颈的预兆，如果此参数值比较高，则可能存在网络瓶颈。
- ❑ `wsrep_last_committed`：此参数代表了全部提交的事务数目。
- ❑ `wsrep_local_cert_failures&wsrep_local_bf_aborts`：此参数代表了事务回滚和检测到的冲突数目。

当集群各个节点的 `wsrep_ready` 选项值均为 ON 时，表明 Galera 集群已经可以正常使用，现在可以在集群上测试 SQL 语句执行情况。为了验证 Galera 集群各个节点之间的数据同步情况，首先在 `controller1-vm` 节点上创建一个名为 `warrior` 的数据库，然后在 `controller2-vm` 和 `controller3-vm` 节点上检查数据库创建语句是否已被执行。

```
//在controller1-vm上创建数据库“warrior”
[root@controller1-vm ~]# MySQL -u root -p
MariaDB [(none)]> create database warrior;
Query OK, 1 row affected (0.10 sec)
//在controller2-vm上查看是否有数据库“warrior”
[root@controller2-vm ~]# MySQL -u root -p
MariaDB [(none)]> show databases;
+-----+
| Database |
+-----+
.....
| warrior |
+-----+
10 rows in set (0.06 sec)
//在controller3-vm上查看是否有数据库“warrior”
[root@controller3-vm ~]# MySQL -u root -p
MariaDB [(none)]> show databases;
+-----+
| Database |
+-----+
.....
| warrior |
```



```
+-----+
10 rows in set (0.37 sec)
```

可以看到，当用户向 controller1-vm 节点发出创建的数据库“warrior”的 SQL 语句后，该 SQL 执行语句会被 wsrep 同步到 controller2-vm 和 controller3-vm 节点上执行，因此在 controller2-vm 和 controller3-vm 节点上也存在“warrior”数据库。同样，如果在 controller3-vm 节点上对“warrior”数据库进行 Drop 操作（可以是任一节点），则 controller1-vm 和 controller2-vm 中也会执行该操作并将“warrior”数据库删除。

```
//在controller3-vm上执行drop操作
[root@controller3-vm ~]# MySQL -u root -p
MariaDB [(none)]> drop database warrior;
Query OK, 0 rows affected (0.02 sec)
//在controller2-vm节点上查看“warrior”是否已被drop
[root@controller2-vm ~]# MySQL -u root -p
MariaDB [(none)]> show databases;
+-----+
| Database |
+-----+
| information_schema |
| cinder |
| glance |
| heat |
| keystone |
| MySQL |
| neutron |
| nova |
| performance_schema |
+-----+
9 rows in set (0.00 sec)
//在controller1-vm节点上查看“warrior”是否已被drop
[root@controller1-vm ~]# MySQL -u root -p
MariaDB [(none)]> show databases;
+-----+
| Database |
+-----+
| information_schema |
| cinder |
| glance |
| heat |
| keystone |
| MySQL |
| neutron |
| nova |
| performance_schema |
+-----+
9 rows in set (0.30 sec)
```

当用户在 controller3-vm 节点上将“warrior”数据库删除后，controller1-vm 和 con-

troller2-vm 节点上的“warrior”数据库也被自动删除。上述两个事例说明，在 Galera 集群中，客户端向任何一个节点发起的 SQL 操作语句均会被同步到其他节点并被执行，从而保证所有节点的数据一致性。在验证了 Galera 集群的数据一致性后，现在来验证 Galera 集群的高可用性（准确地说，是集成了 HAProxy 和 Pacemaker 的 Galera 集群），假设 controller2-vm 节点故障（关闭 controller2-vm 电源），则可以通过观察 controller1-vm 或者 controller3-vm 节点上的 Galera 集群运行状态来判断此时集群是否可用。

//controller2-vm 宕机后，在 controller3-vm 中查看 Galera 集群状态

```
[root@controller3-vm ~]# MySQL -u root -p
MariaDB [(none)]> show status like "wsrep_%";
```

Variable_name	Value
wsrep_incoming_addresses	192.168.142.110:3306,192.168.142.112:3306
wsrep_Cluster_conf_id	11
wsrep_Cluster_size	2
wsrep_Cluster_state_uuid	3c02a77e-e2d4-11e5-af4e-623059392f54
wsrep_Cluster_status	Primary
wsrep_connected	ON
wsrep_ready	ON
wsrep_thread_count	2

48 rows in set (0.01 sec)

注意观察此时集群中的 wsrep_Cluster_size 和 wsrep_incoming_addresses 选项值，这里 wsrep_Cluster_size 为 2，wsrep_incoming_addresses 中也只有 controller1-vm 和 controller3-vm 节点 IP 地址，即 controller2-vm 节点故障后，集群自动将其移除。此外，注意观察表明集群是否可用的 wsrep_ready 选项是否仍然为 ON。这里 wsrep_ready 确实仍然为 ON，说明 controller2-vm 节点宕机后 GaleraCluster 仍然可以正常使用。

假设 controller2-vm 节点故障仍未恢复，controller3-vm 节点也故障，即此时的 Galera Cluster 中仅有 controller1-vm 节点正常运行，根据 Galera 集群的高可用性，此时的集群应该仍然正常运行并可以对外提供数据库服务，即仍然可以正常执行 SQL 操作语句。

//在 controller1-vm 上查看集群状态是否正常

```
[root@controller1-vm ~]# MySQL -u root -p
MariaDB [(none)]> show status like "wsrep_%";
```

Variable_name	Value
wsrep_incoming_addresses	192.168.142.110:3306
wsrep_Cluster_conf_id	12

```

| wsrep_Cluster_size           | 1 |
| wsrep_Cluster_state_uuid    | 3c02a77e-e2d4-11e5-af4e-623059392f54 |
| wsrep_Cluster_status       | Primary |
| wsrep_connected            | ON |
.....
| wsrep_ready                 | ON |
| wsrep_thread_count         | 2 |
+-----+

```

48 rows in set (0.01 sec)

//在集群中创建数据库“shutdown_two_node”

```
MariaDB [(none)]> create database shutdown_two_node;
```

Query OK, 1 row affected (0.03 sec)

//查看数据库创建语句执行结果

```
MariaDB [(none)]> show databases;
```

```

+-----+
| Database |
+-----+
.....

```

```
| shutdown_two_node |
```

```
+-----+
```

10 rows in set (0.00 sec)

从 controller1-vm 节点的集群状态输出情况来看，此时集群中只有一个节点（wsrep_Cluster_size 为 1），但是集群状态仍然正常（wsrep_ready 为 ON）。此外，还在集群中执行数据库创建语句“create database shutdown_two_node”创建了一个名为 shutdown_two_node 的数据库。现在，重新启动 controller2-vm 和 controller3-vm 节点，验证这两个节点是否自动加入 Galera 集群，并重新同步节点故障期间 Galera 集群进行的 SQL 操作语句，如果自动加入且同步成功，应该可以在 controller2-vm 和 controller3-vm 节点中看到在其故障期间创建的名为 shutdown_two_node 的数据库。

//查看controller2-vm节点是否存在数据库“shutdown_two_node”

```
[root@controller2-vm ~]# MySQL -u root -p
```

```
MariaDB [(none)]> show databases;
```

```

+-----+
| Database |
+-----+
.....

```

```
| shutdown_two_node |
```

```
+-----+
```

10 rows in set (0.20 sec)

//查看controller3-vm节点是否存在数据库“shutdown_two_node”

```
[root@controller3-vm ~]# MySQL -u root -p
```

Enter password:

```
MariaDB [(none)]> show databases;
```

```

+-----+
| Database |
+-----+
.....

```

```
| shutdown_two_node |
+-----+
10 rows in set (0.07 sec)
```

可以看到, 当 controller2-vm 和 controller3-vm 节点均宕机后, 虽然集群中只有 controller1-vm 一个正常运行的节点, 但是 Galera 集群仍然正常运行。在仅有 controller1-vm 节点运行的 Galera 集群中创建一个名为 shutdown_two_node 的数据库, 当之前宕机的两个节点重新启动之后, 两个节点均自动重新加入 Galera 集群中, 并自动同步故障期间的数据库创建语句。上述验证结果表明, 基于 HAproxy 和 Pacemaker 的 MariaDB Galera 集群确实实现了 MariaDB 数据库服务的高可用性, 同时也表明 Galera 集群在多个节点故障的特殊情况下, 仍然可以保证集群节点之间的数据同步和数据一致性。

7.2 非关系型数据库——MongoDB

7.2.1 NoSQL 概述

从历史溯源来看, NoSQL 一词首先由 Carlo Strozzi 于 1998 年提出, 最初 “NoSQL” 这个组合词是用来指代 Carlo Strozzi 开发的一款没有 SQL 功能的轻量级开源关系型数据库, 这与现在对 NoSQL 的通俗定义是有很大区别的。但是, 随着技术的发展, NoSQL 的定义也出现了顺应技术革新的改变, Carlo Strozzi 也认为其实真正需要的不仅仅是 “NO SQL”, 而应该是 “No Relational”, 即不是没有 SQL 功能的数据库, 而应该是非关系型数据库。

2009 年初, 来自音乐电台公司 Last.fm 的 Johan Oskarsson 发起了一场关于开源分布式数据库的讨论。在这次讨论中, NoSQL 一词被 Eric Evans 再次提出, 用来指代一些刚出现和兴起的非关系型、分布式并且不遵循关系型数据库 ACID 原则 (即原子性、一致性、独立性和持久性) 的新型数据储存系统。当然, Eric Evans 在当时的讨论会中使用 NoSQL 这个词并不仅仅是因为其字面意思, 当时很多经典的关系型数据库名字都叫 xxSQL, 如 MySQL、MS SQL、PostgreSQL 等, 所以为了表示跟这些关系型数据库在功能使用和定位上的不同, 就采用了 NoSQL 来指代这类新型数据库。

仍然是在 2009 年, 在亚特兰大举行的 “No:SQL(East)” 讨论会可以认为是 NoSQL 类型数据库的一个里程碑, 讨论会的口号是 “select fun, profit from real_world where relational=false;”。从这句口号可以看出, NoSQL 的支持和开发者对 NoSQL 的解释仍然还是聚焦在 “No Relational”, 即非关系型上, 强调的是 Key-Value 键值对的存储和面向文档数据库的优点, 而不仅仅是反对关系型数据库管理系统 (RDBMS)。

从历史发展来看, 2009 年是 NoSQL 发展最为炙热的一年, 从此之后, 各种开源的 NoSQL 类型数据库不断涌现并成熟起来, 如今, NoSQL 被普遍认为是 Not Only SQL 的缩写, 即此类型数据库是非关系型的数据库, 主要以 Key-Value 形式存储数据, 同时在数

数据库管理系统上与关系型数据库截然不同。NoSQL 无须遵循 SQL 标准、ACID 原则、表结构等，这类数据库具有非关系型、分布式、开源、水平可扩展等属性。NoSQL 没有模式，它的数据存储格式可以是松散的，所以无须像关系型数据库一样必须预先定义表及其属性（比如表有哪些列，每一列的数据类型是什么等），才能将符合此表属性的数据存储到表中。

当然，NoSQL 类型数据库之所以能够蓬勃发展，主要还是得利于互联网技术在各个领域的渗透和迅速发展，关于为何要使用 NoSQL 数据库和 NoSQL 数据库得以快速发展的原因，Robin 做出了如下的“三高”总结^①。

（1）High Performance

即对数据库高并发读写的需求。Web2.0 网站要求根据用户个性化信息来实时生成动态页面和提供动态信息，所以基本上无法使用动态页面静态化技术，因此数据库并发负载非常高，往往要达到每秒上万次读写请求。关系数据库应付上万次 SQL 查询还勉强顶得住，但是应付上万次 SQL 写数据请求，硬盘 I/O 就已经无法承受了。其实对于普通的 BBS 网站，往往也存在对高并发写请求的需求，例如像 JavaEye 网站的实时统计在线用户状态，记录热门帖子的点击次数，投票计数等，因此这是一个相当普遍的需求。

（2）Huge Storage

即对海量数据高效率存储和访问的需求。类似 Facebook、Twitter、Friendfeed 这样的 SNS 网站，每天用户产生海量的用户动态，以 Friendfeed 为例，一个月就达到了 2.5 亿条用户动态，对于关系数据库来说，在一张 2.5 亿条记录的表里面进行 SQL 查询，效率是极低乃至不可忍受的。再例如大型 Web 网站的用户登录系统，例如腾讯、盛大等游戏网站门户，动辄数以亿计的账号，单纯的关系数据库也很难应付。

（3）High Scalability & High Availability

即对数据库的高可扩展性和高可用性的需求。在基于 Web 的架构中，数据库是最难进行横向扩展的，当一个应用系统的用户量和访问量与日俱增时，用户的数据库却没有办法像 Web Server 和 App Server 一样简单地通过添加硬件和服务节点来扩展性能和负载能力。所以对于很多需要提供 24 小时不间断服务的网站来说，对数据库系统进行升级和扩展是非常痛苦的事情，往往需要停机维护和数据迁移，但非关系型数据库可以很好地满足这种需求。

传统的关系数据库在面对上述“三高”应用场景时显得力不从心，为了解决这类“三高”的问题，非关系数据库应运而生。到目前为止，各种各样的非关系数据库，尤其是键值数据库（Key-Value DB）更是有百家争鸣之势。目前在各个领域和公司使用的开源非关系数据库已经不少于 10 种，例如 MongoDB、Redis、Tokyo Cabinet、Cassandra、Voldemort、Dynomite、HBase、CouchDB、Hypertable、Riak、Tin、Flare、Lightcloud、KiokuDB、

① <http://robbin.iteye.com/blog/524977>

Scalaris、Kai 和 ThruDB 等。当然这些数据库各有各的优点和特性，但又具备一定的共性特征，这些共性特征可以归纳如下：

- ❑ 代表着不仅仅是 SQL。
- ❑ 没有声明性查询语言。
- ❑ 没有预定义的模式。
- ❑ Key-Value 键值对存储，列存储，文档存储，图形数据存储。
- ❑ 最终一致性，而非 ACID 属性。
- ❑ 非结构化和不可预知的数据。
- ❑ 符合 CAP 定理。
- ❑ 高性能、高可用性和可伸缩性。

在各种 NoSQL 数据库中，每一种或者几种非关系型数据库又是针对特定的数据存取机制而开发的，它们在数据存储和使用上又都有比较适合自己功能的领域，表 7-1 是根据数据存储类型对各种 NoSQL 数据库进行的归纳分类。

表 7-1 NoSQL 数据库分类

数据存储类型	NoSQL 数据库代表	特 点
列存储	Hbase Cassandra Hypertable	按列存储数据，最大的特点是方便存储结构化和半结构化数据，方便做数据压缩，对某一列或者某几列的查询有非常大的 IO 优势
文档存储	MongoDB CouchDB	文档存储一般用类似 JSON 的格式存储，存储的内容是文档型。这样也就可以对某些字段建立索引，实现关系数据库的某些功能
Key-Value 存储	TokyoCabinet/Tyrant BerkeleyDB MemcacheDB Redis	可以通过 Key 快速查询到其 Value。一般来说，存储数据不会管 Value 的格式，只要是客户端提交的数据就全部存储
图形存储	Neo4J FlockDB	图形关系的最佳存储。若用传统关系数据库则性能低下，而且设计使用不方便
对象存储	db4o Versant	通过类似面向对象语言的语法操作数据库，通过对象的方式存取数据
XML 数据库	BerkeleyDB XML BaseX	高效的存储 XML 数据，并支持 XML 的内部查询语法，比如 XQuery、XPath

7.2.2 MongoDB 概述

MongoDB(<https://www.MongoDB.com>) 是介于关系数据库和非关系数据库之间的一种数据库开源软件，是非关系数据库中功能最丰富、最像关系数据库的 NoSQL 数据库，其名称源自“Humongous”，即海量的意思。MongoDB 是由 10gen 开发并维护的一种开源、高性能、可扩展、无模式、面向文档（Document-Oriented）的数据库，其内部存储的是一种类 JSON(JSON-Like) 的结构化数据。MongoDB 是一个面向文档的数据库系统，使用 C++

编写, 不支持 SQL, 但有自己功能强大的查询语法。MongoDB 使用 BSON(Binary-JSON) 作为数据存储和传输的格式 (BSON 是一种类似 JSON 的二进制序列化文档, 支持嵌套对象和数组)。MongoDB 最大的特点是支持的查询语言非常强大, 其语法类似于面向对象的查询语言, 几乎可以实现类似关系数据库单表查询的绝大部分功能, 而且还支持对数据建立索引, 存储数据非常方便。MongoDB 作为一种 NoSQL 类型的数据库, 具有如下功能特征。

- ❑ 面向文档与集合的存储, 适合存储对象类型数据。
- ❑ 模式自由 (Schema-Free)。
- ❑ 支持动态查询。
- ❑ 支持完全索引, 包含内部对象。
- ❑ 支持复制和故障恢复。
- ❑ 使用高效的二进制数据存储, 包括大型对象, 如视频等大对象数据。
- ❑ 自动处理碎片, 易于扩展以支持云计算层次的扩展性。
- ❑ 支持 Ruby、Python、Java、C++ 和 PHP 等多种语言。
- ❑ 文件存储格式为 BSON (一种类 JSON 的扩展格式)。
- ❑ 可通过网络访问数据库。

在实际使用中, 并非所有数据存储场景都适合使用 MongoDB 数据库, 根据其特性, 最适合使用 MongoDB 的场景有以下几种。

- ❑ 对数据库的伸缩性要求较高的场景: MongoDB 非常适合由数十或数百台服务器组成的数据库集群, MongoDB 数据库集群极易进行横向水平扩展。
- ❑ 需要海量数据存储和查询的场景: 使用传统的关系数据库海量存储某些非结构化的数据时会显著增加成本, 选择 MongoDB 则可以大幅度降低数据存储成本, 并且查询效率会提高很多。
- ❑ 进行对象数据和 JSON 格式数据存储的场景: MongoDB 的 BSON 数据格式非常适合文档格式化的存储及查询。

MongoDB 是一种面向文档和数据集合的数据库, 注意这里提到面向文档中的文档并不是指文件, 而是指存储的数据是以文档为最小单位, 并且文档是由 Key-Value 构成的存储单元, Key 是字符串, 而 Value 可以是任意格式的数据。在 MongoDB 中, 每个文档相当于关系数据库中的一条记录, 例如下面 JSON 格式的 Key-Value 键值对就是 MongoDB 中的一个文档, 也是 MongoDB 的最小存储单元。

```
{ "name": "sjx", "alias": "warrior" }
```

MongoDB 中面向数据集合的特性指明数据被分组存储在不同的集合中, 一个集合中包含有若干个文档, 即集合是由文档组成的, 每个集合在数据库中都有一个唯一的标识名。集合的概念类似关系型数据库里的表, 不同的是它不需要定义任何模式, 即模式自由。模式自由即集合里面没有列和行这种关系数据库才有的概念。MongoDB 以 BSON 的格式存储

数据，BSON 存储意味着存储在集合中的文档是以 Key-Value 值对的形式存储的，Key 用于唯一标识一个文档，为字符串类型，而 Value 则可以是各种复杂的数据类型。MongoDB 很像 MySQL，Document 对应 MySQL 的 ROW，Collection 对应 MySQL 的 Table，如图 7-9 所示。

在图 7-9 中，MongoDB 的最小存储单位就是文档对象（JSON 格式），对应于关系型数据库的行，数据在 MongoDB 中以 BSON 文档的格式存储在磁盘上。文档由 Key-Value 键值对组成，同时键值对是有序的，若两个文档对象顺序不相等，则两个存储对象不能对等，如图 7-10 所示。

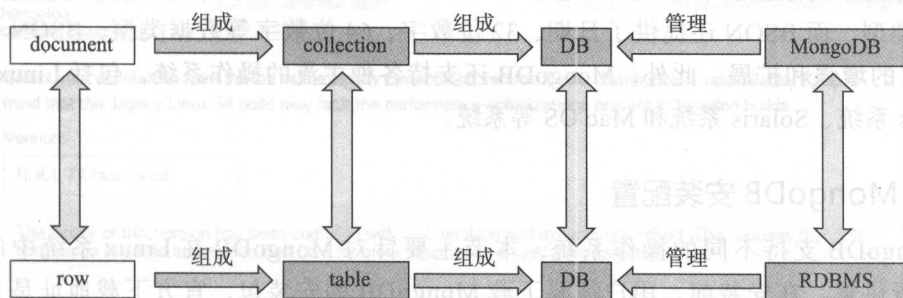


图 7-9 MongoDB 与 MySQL 数据库概念对比

MongoDB 中的文档集合称为 Collection，对应着关系数据库的 Table，与表不同的是，Table 有自己的模式（Schema），而集合是无模式的（Schema-Free），如图 7-11 所示。

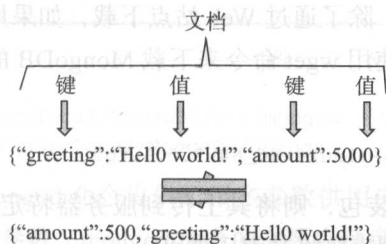


图 7-10 MongoDB 中的有序 Key-Value 值对



图 7-11 MongoDB 的无模式 Collection

在 MongoDB 内部，每个数据库都会包含一个 .ns 为后缀的文件和一些数据存储文件，而且这些数据存储文件会随着数据量的增加而变得越来越多。如果系统中有一个叫作 local 的 MongoDB 数据库，那么构成 local 这个数据库的文件就会由 local.ns、local.0、local.1 等组成（数据存储文件的后缀以整数递增）。以 CentOS7 系统为例，MongoDB 安装之后，默认数据库存放路径为 /var/lib/MongoDB，假设 MongoDB 中有一个名为 local 的数据库，则可以看到该目录内容如下。

```
[root@controller2-vm ~]# cd /var/lib/MongoDB
[root@controller2-vm MongoDB]# ll
```

```
total 1162768
-rw----- 1 MongoDBMongoDB 16777216 Apr 15 23:45 ceilometer.0
-rw----- 1 MongoDBMongoDB 16777216 Apr 15 23:45 ceilometer.ns
drwxr-xr-x 2 MongoDBMongoDB      61 May  8 10:58 journal
-rw----- 1 MongoDBMongoDB 67108864 May  6 07:05 local.0
-rw----- 1 MongoDBMongoDB 536608768 May  8 10:58 local.1
-rw----- 1 MongoDBMongoDB 536608768 May  8 10:58 local.2
-rw----- 1 MongoDBMongoDB 16777216 May  8 10:58 local.ns
-rwxr-xr-x 1 MongoDBMongoDB      6 May  6 07:05 mongod.lock
```

MongoDB 的文档存储对象使用 BSON 来组织数据，BSON 类似于 JSON，但是 JSON 只是一种简单的数据表示方式，只包含了 null、布尔、数字、字符串、数组及对象一共 6 种数据类型，而 BSON 还提供了日期、32 位数字、64 位数字等数据类型，BSON 可以看作 JSON 的增强和扩展。此外，MongoDB 还支持各种主流的操作系统，包括 Linux 系统、Windows 系统、Solaris 系统和 Mac OS 等系统。

7.2.3 MongoDB 安装配置

MongoDB 支持不同的操作系统，本节主要针对 MongoDB 在 Linux 系统中的安装配置进行讲解。在安装前，用户需要下载 MongoDB 的安装包，官方下载地址是 <https://www.MongoDB.com/download-center?jmp=nav#community>。登录安装包下载页面后，用户可以根据需要选择不同的操作系统类型，并选择操作系统的版本进行下载。以 Redhat 发行的 Linux 操作系统为例，系统版本为 RHEL7，则 MongoDB 的下载页面如图 7-12 所示。

图 7-12 中下载的 MongoDB 是一个归档压缩包，除了通过 Web 站点下载，如果用户系统外网连接不受限制，也可以直接在 Linux 系统中使用 `wget` 命令来下载 MongoDB 的归档压缩包，如下：

```
wget https://fastdl.MongoDB.org/linux/MongoDB-linux-x86_64-3.0.6.tgz
```

如果是通过 Web 站点下载的 MongoDB 归档安装包，则将其上传到服务器特定目录，如果通过 `wget` 命令下载的 MongoDB 归档压缩包，则直接将其解压即可：

```
tar -zxvf MongoDB-linux-x86_64-3.0.6.tgz
```

在正式安装 MongoDB 之前，可以先将解压后的 MongoDB 安装包拷贝到指定目录：

```
mv MongoDB-linux-x86_64-3.0.6/ /usr/local/MongoDB
```

然后创建自定义的数据库存放目录文件夹，即数据库存放位置和日志存放目录：

```
mkdir /usr/local/MongoDB/data
mkdir /usr/local/MongoDB/dblogs
```

这里，将 MongoDB 的数据库文件存入 `/usr/local/MongoDB/data` 目录，将日志文件存入 `/usr/local/MongoDB/dblogs` 目录。由于 MongoDB 没有具体的安装和配置过程，即只需

解压便可启动运行，因而 MongoDB 的安装非常简单。解压完成之后，通过 `mongod` 命令即可启动 MongoDB 服务，启动过程中通常只需指定数据库路径和日志路径即可：

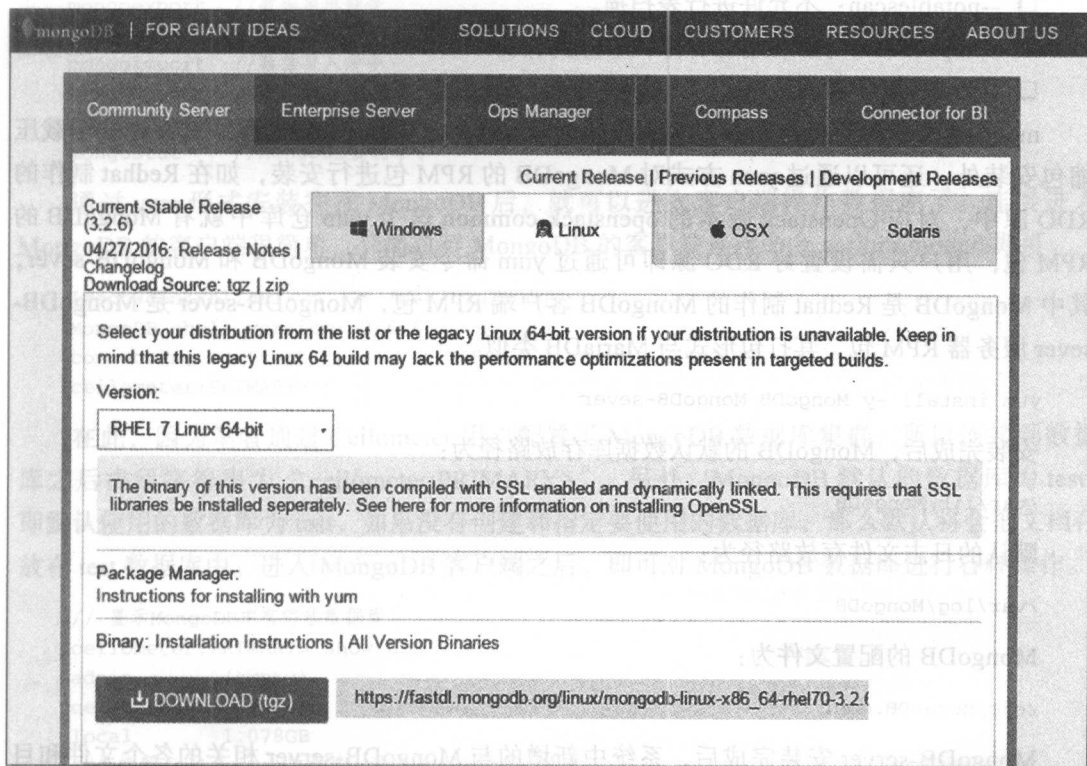


图 7-12 MongoDB 安装包下载页面

```
/usr/local/MongoDB/bin/mongod --dbpath=/usr/local/MongoDB/data --fork --logpath=/usr/local/MongoDB/dblogs
```

`mongod` 命令提供了很多参数供用户选择，用户可以根据需求在启动 MongoDB 时指定相应的参数，`mongod` 命令的主要启动参数解释如下：

- ❑ `--dbpath`：用以指定数据库的存放目录。
- ❑ `--port`：用以指定数据库的端口，默认是 27017。
- ❑ `--bind_ip`：绑定 MongoDB 的监听 IP 地址。
- ❑ `--directoryperdb`：为每个数据库创建一个独立子目录。
- ❑ `--logpath`：指定日志存放目录。
- ❑ `--logappend`：指定日志生成方式（追加 / 覆盖）。
- ❑ `--pidfilepath`：指定进程文件路径，如果不指定，将不会产生进程文件。
- ❑ `--KeyFile`：集群模式的关键标识。
- ❑ `--journal`：启用日志。

❑ `--nssize`: 指定 `.ns` 文件的大小, 单位为 MB, 默认是 16MB, 最大是 2GB。

❑ `--maxConns`: 指定最大的并发连接数。

❑ `--notablescan`: 不允许进行表扫描。

❑ `--noprealloc`: 关闭数据文件的预分配功能。

❑ `--fork`: 以后台 Daemon 形式运行 MongoDB 服务。

`mongod` 命令的更多启动参数选项可以通过 `mongod --help` 查看。除了官方网站下载压缩包安装外, 还可以通过 `yum` 方式对 MongoDB 的 RPM 包进行安装, 如在 Redhat 制作的 RDO 源中, 对应 Openstack 版本的 `openstack-common` 这个 `yum` 仓库中就有 MongoDB 的 RPM 包, 用户只需设置好 RDO 源即可通过 `yum` 命令安装 MongoDB 和 `MongoDB-sever`, 其中 MongoDB 是 Redhat 制作的 MongoDB 客户端 RPM 包, `MongoDB-sever` 是 MongoDB-sever 服务器 RPM 包, 其打包形式与 MariaDB 类似。

```
yum install -y MongoDB MongoDB-sever
```

安装完成后, MongoDB 的默认数据库存放路径为:

```
/var/lib/MongoDB
```

默认的日志文件存放路径为:

```
/var/log/MongoDB
```

MongoDB 的配置文件为:

```
/etc/MongoDB.conf
```

MongoDB-server 安装完成后, 系统中新增的与 MongoDB-server 相关的各个文件和目录如下:

```
[root@controller1-vm ~]# rpm -ql MongoDB-server
/etc/logrotate.d/MongoDB
/etc/MongoDB-shard.conf
/etc/MongoDB.conf
/etc/sysconfig/mongod
.....
```

MongoDB 安装完成后, 在 `/usr/bin` 目录下存放了很多与 MongoDB 相关的可执行命令文件, 用户可以在该目录下找到全部的 MongoDB 命令。

```
[root@controller1-vm ~]# rpm -ql MongoDB
/usr/bin/bsondump
/usr/bin/mongo
/usr/bin/mongodump
.....
```

MongoDB 提供了很多目录用以访问控制和操作, 其中比较重要的几个数据库操作命令的功能解释如下:

```

mongo           //客户端命令,用以连接MongoDB
mongod          //服务器端命令,用以启动MongoDB
mongodump       //MongoDB备份命令
mongoexport     //数据导出命令
mongoimport     //数据导入命令
mongorestore    //数据恢复命令
mongos          //数据分片程序命令,支持数据的横向扩展
mongostat       //MongoDB监视命令

```

通过 yum 形式安装完成 MongoDB 后,就可以进入客户端操作数据库了。连接进入 MongoDB 的客户端很简单,只需执行 MongoDB 的客户端连接命令 `usr/bin/mongo` 即可。

```

[root@controller1-vm ~]# /usr/bin/mongo
MongoDB shell version: 2.6.6
connecting to: test
ceilometer:PRIMARY>

```

在此,因为笔者通过 Ceilometer 用户配置了 MongoDB 数据库集群,所以连接到数据库之后提示字符串为“`ceilometer:PRIMARY>`”。另外, MongoDB 默认的数据库为 `test`,即默认使用的数据库为 `test`,如果没有创建和指定要使用的数据库,那么默认将会把文档存放在 `test` 数据库中。进入 MongoDB 客户端之后,即可对 MongoDB 数据库进行各种操作。

```

// 显示MongoDB中有哪些数据库
ceilometer:PRIMARY> show dbs
admin           (empty)
ceilometer      0.031GB
local           1.078GB
//切换到ceilometer数据库,这与MySQL语法一样
ceilometer:PRIMARY> use ceilometer
switched to db ceilometer
// 显示当前使用的db名称
ceilometer:PRIMARY> db
ceilometer
// 显示当前db中有哪些collections(集合)
ceilometer:PRIMARY> show collections
alarm
alarm_history
event
meter
resource
system.indexes
//查看可以对集合event进行哪些操作
ceilometer:PRIMARY> db.event.help()
DBCollection help
db.event.find().help() - show DBCursor help
db.event.count()
db.event.copyTo(newColl) - duplicates collection by copying all documents to
newColl; no indexes are copied.

```

```

.....
//切换到test数据库
ceilometer:PRIMARY>use test
//删除当前数据库(当前为test)
ceilometer:PRIMARY> db.dropDatabase()
{ "dropped" : "test", "ok" : 1 }
// 查看删除之后的数据库
ceilometer:PRIMARY> show dbs
admin          (empty)
ceilometer     0.031GB
local          1.078GB
//切换到local数据库
ceilometer:PRIMARY> use local
switched to db local
//查看local数据库中的collections
ceilometer:PRIMARY> show collections
me
oplog.rs
replset.minvalid
slaves
startup_log
system.indexes
system.replset
//查询slaves集合中有哪些文档数据
ceilometer:PRIMARY> db.slaves.find()
{ "_id" : ObjectId("56c45f6f15d3a4d4325eaf78"), "config" : { "_id" : 0, "host" :
    : "controller3-vm:27017" }, "ns" : "local.oplog.rs", "syncedTo" : Timestamp
    (1462676335, 1) }
{ "_id" : ObjectId("56c45ffa7bb1bc87f75d99be"), "config" : { "_id" : 2, "host" :
    : "controller2-vm:27017" }, "ns" : "local.oplog.rs", "syncedTo" : Timestamp
    (1462676335, 1) }
// 退出MongoDB客户端
ceilometer:PRIMARY> exit
bye

```

7.2.4 MongoDB Replica Set 概述

对于生产系统而言，使用的任何数据库都应该具备高可用性。作为 NoSQL 数据库中的杰出代表，MongoDB 在数据库集群和高可用方面已有成熟的解决方案。MongoDB 官方社区推荐的高可用集群方案是 Replication (Replica) Set，即数据集复制。较早版本中的 Master-Slave 复制方案已被 Replication Set 方案代替，而在 MongoDB3.2 版本中，Master-Slave 复制功能已被抛弃，因此，本节仅介绍当前官方社区推荐和支持的 Replication Set 集群高可用方案。Replication Set 是一组维护共同数据集的 mongod 进程实例组合，Replica Set 集群中包含多个负责数据维护的节点和一个可选的仲裁节点，其中，负责数据维护的节点中有且仅有一个节点被选举为 Primary 节点，其余节点都是 Secondary 节点。

在 Replica Set 中，Primary 节点接收来自客户端的全部写请求，并且负责将所有对数

据集进行变更的操作 (oplog) 记录到其日志文件中, 所有 Secondary 节点复制 Primary 节点的 oplog 文件, 并在本地数据库重现 oplog 中记录的操作, 以保证本地数据库与 Primary 节点的数据一致性。如果 Primary 节点宕机, 则 Replica Set 会重新从 Secondary 节点中选举新的 Primary 节点。另外, Primary 节点与 Secondary 节点之间的复制是异步复制, 因此 Secondary 与 Primary 之间的数据可能存在一定的时间差。默认情况下, Secondary 节点不允许读取数据, 但是可以通过客户端的配置使 Secondary 可读, 不过, 由于 Primary 与 Secondary 之间的复制是异步进行的, 如果从 Secondary 节点上读取数据集, 则读取的数据集可能并不是 Primary 节点上更改过的数据, 即有可能在读取 Secondary 节点上的数据时, Primary 上的数据还没有同步到 Secondary 节点上, 因此, 正常情况下不建议访问 Secondary 节点。三节点组成的 Replication Set 集群 (一个 Primary 节点和两个 Secondary 节点) 拓扑如图 7-13 所示。

在 Replication Set 集群中, Arbiter (仲裁) 节点是可选的, 它并不参与数据维护和处理客户端读写请求, 仅在 Primary 节点故障并需要重新选举新的 Primary 时作为仲裁节点发挥作用。通常, 如果 Replication Set 集群中的成员节点数目为偶数, 则需要 Arbiter 节点来参与新 Primary 节点的选举, Arbiter 节点对系统硬件要求很低, 其可以部署到其他应用服务器上, 但是不能将其部署在 Primary 和任何一个 Secondary 节点上, 具有 Arbiter 节点的 Replication Set 集群架构如图 7-14 所示。

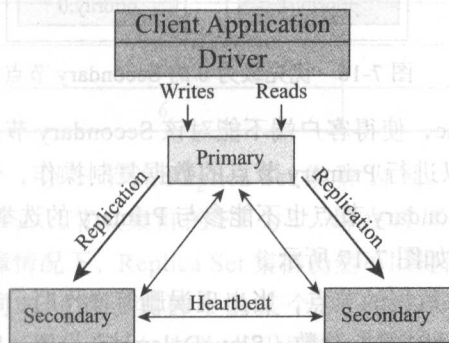


图 7-13 三节点 Replica Set 集群拓扑

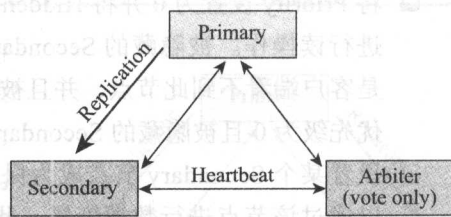


图 7-14 具有仲裁节点的 Replica Set 集群

在 Replica Set 集群中, 当各个 Secondary 节点与 Primary 的通信延时超过 10s 时, 其中的某个候选 Secondary 节点会被选举为新的 Primary 节点。当然, 要成为新的 Primary 节点, 则该 Secondary 节点就必须得到过半的节点票数, 这也是 Replica Set 集群中建议引入仲裁节点的主要原因, 即防止在 Primary 节点选举时出现票数五五平分的情况。在新的 MongoDB3.2 版本中, 引入了新的 Replication 协议, 即 Version1.0; 而以前的 Replication 协议为 version0.0。新的 Replication 协议减少了 Replica Set 集群在 Primary 节点故障时自动 Failover 所消耗的时间, 其 Failover 过程如图 7-15 所示。

在目前的 MongoDB 版本中 (Mongo-DB3.2.6), 一个 Replica Set 集群可以拥有 50 个成

员，但是只有 7 个成员节点能够参与 Primary 的选举，若超过 50 个成员节点，则官方建议使用传统的 Master-Slave 复制形式实现数据高可用。此外，部署一个简单的 Replica Set 至少需要三个节点，可以是一个 Primary，一个 Secondary 和一个 Arbiter 节点，也可以是一个 Primary 和两个 Secondary 节点。在实际部署中，用户可以自定义设置 Replica Set 集群中的 Secondary 节点，使其具有与默认 Secondary 节点不同的行为。用户可以根据实际需求将 Secondary 节点配置为如下三种类型节点。

❑ 将 Secondary 节点的 Priority 设置为 0，使其不参与 Primary 的选举，而只负责数据备份。在实际使用中，通常将这种类型的 Secondary 节点放置到异地数据中心，使其仅作为数据库冷备份节点，优先级为 0 的 Secondary 节点架构拓扑如图 7-16 所示。

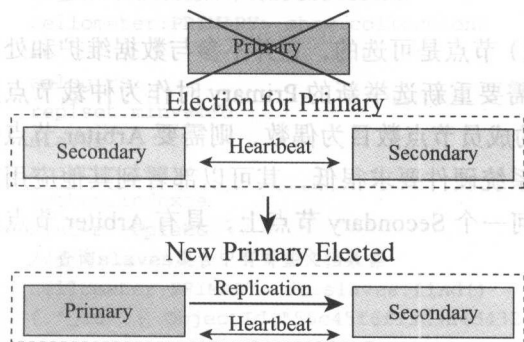


图 7-15 MongoDB Replica Set 集群 Failover

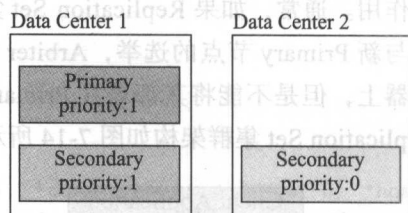


图 7-16 优先级为 0 的 Secondary 节点

❑ 将 Priority 设置为 0 并将 Hidden 设置为 True，使得客户端不能对该 Secondary 节点进行读操作。被隐藏的 Secondary 节点可以进行 Primary 节点的数据复制操作，但是客户端看不到此节点，并且被隐藏的 Secondary 节点也不能参与 Primary 的选举。优先级为 0 且被隐藏的 Secondary 节点拓扑如图 7-17 所示。

❑ 设置某个 Secondary 节点成为保留历史镜像数据的节点，当出现误删等操作时，可以通过该节点进行数据恢复。此功能通过延时保留参数 (SlaveDelay) 来设置，时间单位为秒，通常在开启此功能时，Secondary 节点的优先级应该设为 0，并且 hidden 为 true，即不提供读写访问也不参与 Primary 节点选举，具有延时保留功能的 Secondary 节点拓扑如图 7-18 所示。

7.2.5 MongoDB Replica Set 部署

Replica Set 集群最初的架构设计直接影响到整个数据库集群的容量和性能，官方推荐的标准 Replica Set 部署为三节点部署。三节点部署可以同时提供数据冗余和故障容忍 (Fault Tolerance, FT)，同时也减少了部署的复杂性。此外，目前支持的 Replica Set 集群的最大成员数为 50，可参与 Primary 选举的最大节点数为 7。在集群节点选取方面，

MongoDB 官方社区建议部署奇数个 Replica Set 成员节点，如果是偶数个成员节点，则建议部署一个 Arbiter 节点，因为 Arbiter 节点不进行数据复制，对资源消耗很少，因此 Arbiter 节点也可以部署到其他应用服务器上共享物理资源，而无须单独服务器部署，同时一个 Replica Set 集群中只需部署一个 Arbiter 节点即可。

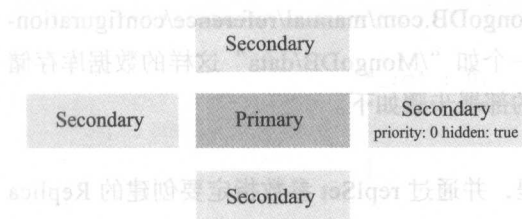


图 7-17 优先级为 0 并被隐藏的 Secondary 节点

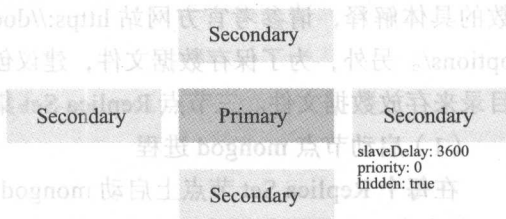


图 7-18 具有延时保留功能的 Secondary 节点

在一个高可用集群中，集群的 FT 是必须考虑的。FT 是一个集群中能够承受的故障节点个数，当集群中故障节点个数超过了 FT 限制时，集群将不能正常工作。表 7-2 是 Replica Set 集群中成员个数与 FT 限制的关系对照表。

表 7-2 节点数目与 FT 限制对照表

Replica Set 成员数目	选举 Primary 需要的成员数目	FT 数目
3	2	1
4	3	1
5	3	2
6	4	2

以三节点 Replica Set 集群为例，如表 7-2 所示，三节点集群的 FT 为 1，即在最坏的两节点故障情况下，Replica Set 集群仍然可以提供正常访问，同时三节点集群中的某个 Secondary 节点要成为 Primary，则在选举过程中其至少要获得两票。为了满足基本的集群功能，本节将部署一个三节点的 Replica Set 集群（这里采用一个 Primary 节点和两个 Secondary 节点的部署方式），其节点拓扑架构如图 7-19 所示。

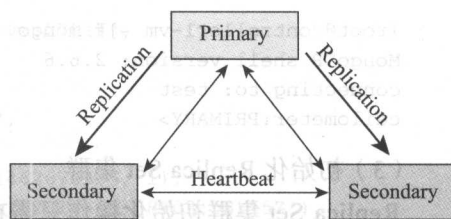


图 7-19 三节点 Replica Set 集群拓扑架构

MongoDB Replica Set 集群部署的整体思路，是先在每个独立节点上安装部署 MongoDB 软件包，并在安装完成后启动该节点上的 mongod 进程，然后再初始化 Replica Set 集群，最后将各个独立的成员节点加入 Replica Set 集群中。对生产环境而言，最好是保证各个 Replica Set 成员节点的独立性，将每一个 mongod 进程部署在独立的机器上。在部署之前，确保每一个节点上都安装了 MongoDB 软件包，并确保成员节点之间的网络通信正

常。另外，在生产环境中，建议将 mongod 进程绑定到 MongoDB 的标准端口 27017 上，同时通过设置 bind_ip 参数指定 mongod 进程的监听 IP 地址。在启动 mongod 进程之前，先根据需求配置好 MongoDB 的配置文件（如设置 bind_ip），配置文件通常是 /etc/mongod.conf。通常情况下，如果没有特殊需求，无须进行过多的配置文件修改。关于配置文件参数的具体解释，请参考官方网站 <https://docs.MongoDB.com/manual/reference/configuration-options/>。另外，为了保存数据文件，建议创建一个如 “/MongoDB/data” 这样的数据库存储目录来存放数据文件。三节点 Replica Set 集群的部署步骤如下。

（1）启动节点 mongod 进程

在每个 Replica Set 节点上启动 mongod 进程，并通过 replSet 参数指定要创建的 Replica Set 的名字，如果客户端应用程序连接到多个 Replica Set，确保每个 Replica Set 具有不同的名字，有些应用程序后端驱动会根据 Replica Set 的名称进行分组连接，如下命令行是在启动 mongod 进程时设置 replSet 名字为 ceilometer。

```
controller1-vm# mongod --replSet=ceilometer
```

除了命令行方式设置 replSet 名字外，还可以通过 MongoDB 的配置文件进行永久设置，即每次 MongoDB 启动时候，都会自动获取配置文件的 replSet 参数值，从而保证每次启动都设置相同的 replSet 名称，简单起见，可以通过如下方式进行永久设置。

```
controller1-vm#echo "replSet=ceilometer" >> /etc/MongoDB.conf
```

（2）进入 mongo shell 客户端

MongoDB 提供了与用户交互式的命令行客户端，即 mongo shell。进入 mongo shell 的方式很简单，如果 mongod 进程监听默认端口 27017，则只需在系统命令行运行 mongo 命令即可进入 MongoDB 的客户端。

```
[root@controller1-vm ~]# mongo
MongoDB shell version: 2.6.6
connecting to: test
ceilometer:PRIMARY>
```

（3）初始化 Replica Set 集群

Replica Set 集群初始化操作只需在某个成员节点上操作。Replica Set 集群初始化函数是 rs.initiate()，用户只需在节点系统命令中执行不带任何参数的 rs.initiate() 函数即可（注意不是 MongoDB 的命令）。初始化的过程将会使用默认的 Replica Set 配置把当前节点初始化为 Replica Set 集群的成员节点。

```
[root@controller1-vm ~]# rs.initiate()
```

（4）验证初始化后的 Replica Set 节点

在节点上执行 rs.initiate() 初始化函数后，当前节点则被初始化成为 Replica Set 节点。初始化操作完成后，需要对其执行结果进行验证。验证操作需要进入 MongoDB 的客户端命

令行界面，并通过 rs.conf() 函数查看当前节点的 Replica Set 配置信息。

```
ceilmeter:PRIMARY> rs.conf()
{
  "_id" : "ceilmeter",
  "version" : 3,
  "members" : [
    {
      "_id" : 0,
      "host" : "controller1-vm:27017"
    }
  ]
}
```

(5) 扩展 Replica Set 集群节点

Replica Set 集群初始化完成后，集群中仅含有运行初始化函数的节点。现在需要将其余两个节点加入到 Replica Set 集群中，新增节点过程很简单，用户只需通过 MongoDB 的客户端连接到 Replica Set 集群的 Primary 节点，使用 rs.add() 方法即可添加新成员节点。此处需要注意的是，如果 mongo shell 连接的是 Secondary 节点，则必须重新连接，只有连接到 Primary 节点才可进行节点添加操作。在 Replica Set 集群成员节点扩展之前，可以通过 rs.status() 方法检查 Replica Set 集群的当前状态。

```
ceilmeter:PRIMARY> rs.status()
{
  "set" : "ceilmeter",
  "date" : ISODate("2016-05-10T13:03:24Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 1,
      "name" : "controller1-vm:27017",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 2730,
      "optime" : Timestamp(1462676335, 1),
      "optimeDate" : ISODate("2016-05-08T02:58:55Z"),
      "electionTime" : Timestamp(1462882719, 1),
      "electionDate" : ISODate("2016-05-10T12:18:39Z"),
      "self" : true
    }
  ],
  "ok" : 1
}
```

在确认连接到的 mongod 进程为 Primary 节点，并且当前 Replica Set 集群状态正常后，将 controller2-vm 和 controller3-vm 节点加入到 Replica Set 集群中。

```
ceilometer:PRIMARY>rs.add("controller2-vm")
ceilometer:PRIMARY>rs.add("controller3-vm")
```

(6) 最后验证 Replica Set 集群状态

全部节点添加完成之后, Replica Set 集群即可正常运行。在三节点的 Replica Set 集群中, 集群将会选举一个 Primary 节点, 同时有两个 Secondary 节点, 使用 `rs.status()` 方法来验证 Replica Set 的集群运行情况。

```
ceilometer:PRIMARY> rs.status()
{
  "set" : "ceilometer",
  "date" : ISODate("2016-05-10T13:03:24Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 0,
      "name" : "controller3-vm:27017",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 2686,
      "optime" : Timestamp(1462676335, 1),
      "optimeDate" : ISODate("2016-05-08T02:58:55Z"),
      "lastHeartbeat" : ISODate("2016-05-10T13:03:23Z"),
      "lastHeartbeatRecv" : ISODate("2016-05-10T13:03:23Z"),
      "pingMs" : 9,
      "syncingTo" : "controller1-vm:27017"
    },
    {
      "_id" : 1,
      "name" : "controller1-vm:27017",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 2730,
      "optime" : Timestamp(1462676335, 1),
      "optimeDate" : ISODate("2016-05-08T02:58:55Z"),
      "electionTime" : Timestamp(1462882719, 1),
      "electionDate" : ISODate("2016-05-10T12:18:39Z"),
      "self" : true
    },
    {
      "_id" : 2,
      "name" : "controller2-vm:27017",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 2686,
      "optime" : Timestamp(1462676335, 1),
```

```

    "optimeDate" : ISODate("2016-05-08T02:58:55Z"),
    "lastHeartbeat" : ISODate("2016-05-10T13:03:23Z"),
    "lastHeartbeatRecv" : ISODate("2016-05-10T13:03:23Z"),
    "pingMs" : 1,
    "syncingTo" : "controller1-vm:27017"
  },
  "ok" : 1
}

```

使用 `rs.conf()` 方法来检查 Replica Set 集群的最终配置情况。

```

ceilometer:PRIMARY> rs.conf()
{
  "_id" : "ceilometer",
  "version" : 3,
  "members" : [
    {
      "_id" : 0,
      "host" : "controller3-vm:27017"
    },
    {
      "_id" : 1,
      "host" : "controller1-vm:27017"
    },
    {
      "_id" : 2,
      "host" : "controller2-vm:27017"
    }
  ]
}

```

至此，三节点 Replica Set 集群已经部署配置完成。从最后的 `rs.status()` 和 `rs.conf()` 函数输出信息中可以看到，在此处的三节点 Replica Set 配置环境中，Replica Set 的集群名称为 `ceilometer`。`ceilometer` 集群中有三个成员节点，分别是 `controller1-vm`、`controller2-vm` 和 `controller3-vm`。其中，`controller1-vm` 是 Primary 节点，`controller2-vm` 和 `controller3-vm` 是 Secondary 节点。

7.3 本章小结

数据库是任何生产应用系统都不可缺失的组成部分，在其漫长的发展历程中，关系型数据库一直占据统治地位，其中更是涌现出了很多著名的商业数据库软件及开发公司。随着开源精神的普及和开源社区的壮大，各种开源软件不断出现，这其中便有包括像 MySQL 这样伟大的开源数据库。然而随着很多商业软件公司的业务扩展和对成功开源软件的觊觎，开源社区又分支出了像 MariaDB 这样的数据库，并获得社区和用户的极大拥护。在关系型

OpenStack 计算服务

作为 OpenStack 最早也最成熟的项目，计算服务 Nova 从成立至今一直是 OpenStack 最核心的服务。尽管 Nova 服务的部分核心组件在后来的发展中独立成为其他核心服务，如 Nova-volume 已独立成为 Cinder 项目，而 Nova-network 也独立成为 Neutron 项目，但是 Nova 的核心地位和用户部署率仍然是社区最高的项目。OpenStack 作为一种 IaaS 云计算服务，其核心服务资源便是计算、存储和网络，而 Nova 正是计算服务的首要提供者，因此了解并精通 Nova 项目是真正掌握 OpenStack 云计算的关键。此外，在 OpenStack 的生产环境部署中，尤其是私有云部署环境中，Nova 实例的高可用一直是 OpenStack 所缺少的部分，也是阻碍 OpenStack 进入用户生产环境最大的障碍。本章将对 OpenStack 项目及其 Nova 服务进行深入介绍和分析，并详细阐述 Nova 服务项目的核心组件的工作原理和 Nova 实例高可用实现。同时，本章内容也是实现 OpenStack 高可用集群部署的关键，是 OpenStack 高可用集群中虚拟机实例高可用实现的基础。

8.1 OpenStack 项目概述

8.1.1 OpenStack 项目概要

OpenStack 是一个支持公有云、私有云和混合云等各种云计算类型的开源云计算平台项目，其主要目的是简化云计算的实施部署、提供海量扩容的云计算平台和丰富完善的云计算功能集。OpenStack 官方网站对 OpenStack 的介绍如下：OpenStack 是用于控制数据中心计算、存储和网络资源池的云操作系统，在 OpenStack 中，云管理员对资源池的全部管理和控制都可以通过 Dashboard 仪表板实现，同时通过 Web 接口界面实现对用户资源的

供给和控制，官方社区对 OpenStack 项目的抽象软件架构如图 8-1 所示。从 OpenStack 官方社区的定义和图 8-1 中可以看出，OpenStack 的核心项目主要是计算服务、网络服务和存储服务，并通过 Dashboard 将这三大大服务呈现给用户，实现云用户与数据中心资源池的交互。

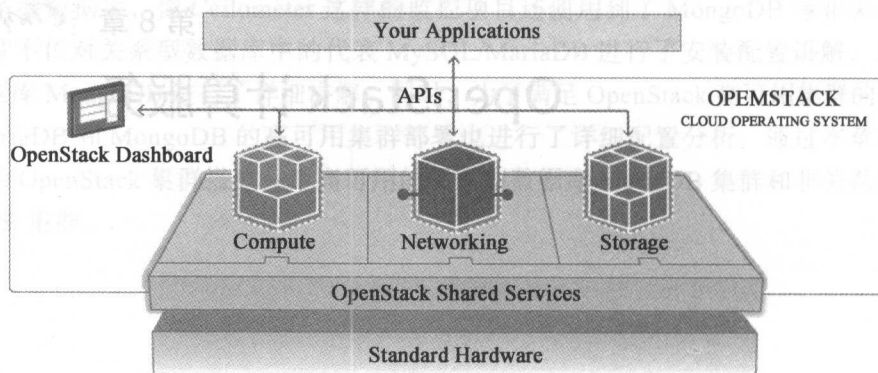


图 8-1 OpenStack 抽象结构图

除了计算（Nova）、存储（Cinder/Swift）和网络（Neutron）三大关键服务，目前 OpenStack 的核心服务项目还包括认证服务（Identity）、控制面板服务（Horizon）和镜像服务（Glance）。OpenStack 七大核心服务项目之间的相互关系和逻辑架构如图 8-2 所示。其中，功能最为复杂和关键的仍然还是 Nova、Neutron 和 Swift 项目，除了这三大项目，Cinder 负责块存储功能，Horizon 负责 Web 界面和 GUI 交互功能，Keystone 负责与 OpenStack 相关的所有项目和用户的身份验证功能。

从架构设计和功能模块来看，OpenStack 是允许用户开发和管理数据中心云基础架构资源的一系列开源软件项目集合。从历史溯源来看，OpenStack 项目最初由 Rackspace 和 NASA 于 2010 年 7 月联合以 Apache 许可证开源发布，最初的 OpenStack 项目仅由 Rackspace 贡献的对象存储 Swift 和 NASA 贡献的计算项目 Nova 组成，第一个版本代号以 RackSpace 所在的美国德州（Texas）首府 Austin 命名。OpenStack 项目每隔半年便会召开一次峰会并在峰会上发布一个全新的版本，同时规划下一次峰会和发行版本，OpenStack 版本代号以 26 个英文字母为首字母从 A 到 Z 的顺序命名，版本代号名称通常是峰会召开地的著名地点名称。

8.1.2 OpenStack 版本发行

在近七年的时间里，OpenStack 社区已成为仅次于 Linux 的最大开源社区，并成为公认成长最快的开源社区和最大的开源云计算社区。从 2010 年 7 月发布的第一个版本 Austin 到 2016 年 4 月重回 OpenStack 发源地 Austin 发布版本 Mitaka，OpenStack 已经历了 13 个版本更迭，并且第 15 个版本 Ocata 已于 2017 年 2 月发布。OpenStack 版本的发行历史和未来

即将发行的版本规划如表 8-1 所示^①，表中的 TBD 表示“待定”（即还未最终确定），EOL 表示“生命终止”（即社区不再维护这个版本）。从表 8-1 中可以看到，当前社区支持的最新稳定版本是 Ocata，正在开发中的版本是 Pike，并计划于 2017 年 9 月初发布。

表 8-1 OpenStack 版本发行统计

代 号	当前状态	最初发行日期	终止日期
Queens	Future	TBD	TBD
Pike	Under Development	2017/09/01	TBD
Ocata	Phase I – Latest release	2017/02/22	TBD
Newton	Phase II – Maintained release	2016/10/6	TBD
Mitaka	Phase III – Legacy release	2016/4/7	2017/4/10
Liberty	EOL	2015/10/15	2016/11/17
Kilo	EOL	2015/4/30	2016/5/2
Juno	EOL	2014/10/16	2015/12/7
Icehouse	EOL	2014/4/17	2015/7/2
Havana	EOL	2013/10/17	2014/9/30
Grizzly	EOL	2013/4/4	2014/3/29
Folsom	EOL	2012/9/27	2013/11/19
Essex	EOL	2012/4/5	2013/5/6
Diablo	EOL	2011/9/22	2013/5/6
Cactus	Deprecated	2011/4/15	—
Bexar	Deprecated	2011/2/3	—
Austin	Deprecated	2010/10/21	—

根据 OpenStack Austin 峰会的统计，OpenStack 每年两次的峰会参与人数已由成立之初的 75 人发展到 2016 年重回 Austin 峰会时的 7500 人，而 OpenStack 的项目组成已由最初的 Swift 和 Nova，演变到如今的 Nova、Swift、Neutron、Cinder、Keystone、Glance 和 Dashboard 七大核心服务，以及 Ceilometer、Heat、Trove、Sahara、Magnum、Ironi、Zaqar 和 Congress 等十二个可选项目，尤其是随着东京峰会时 OpenStack 第 12 个版本 Liberty 的发布，OpenStack 社区正式进入大帐篷（Big Tent）模式，截止 Ocata 版本发布时，被社区正式接受发行的 OpenStack 项目已超过 50 个。表 8-2 为 2017 年 2 月 OpenStack 的 Ocata 版本发布时对应发行的 OpenStack 服务项目版本和对应服务项目的目前最近更新版本，其中的“最早版本”列为伴随 Ocata 版本发行时的服务项目版本，而“最近更新版本”为基于最早发行版本的更新版本。

① <https://releases.openstack.org/>

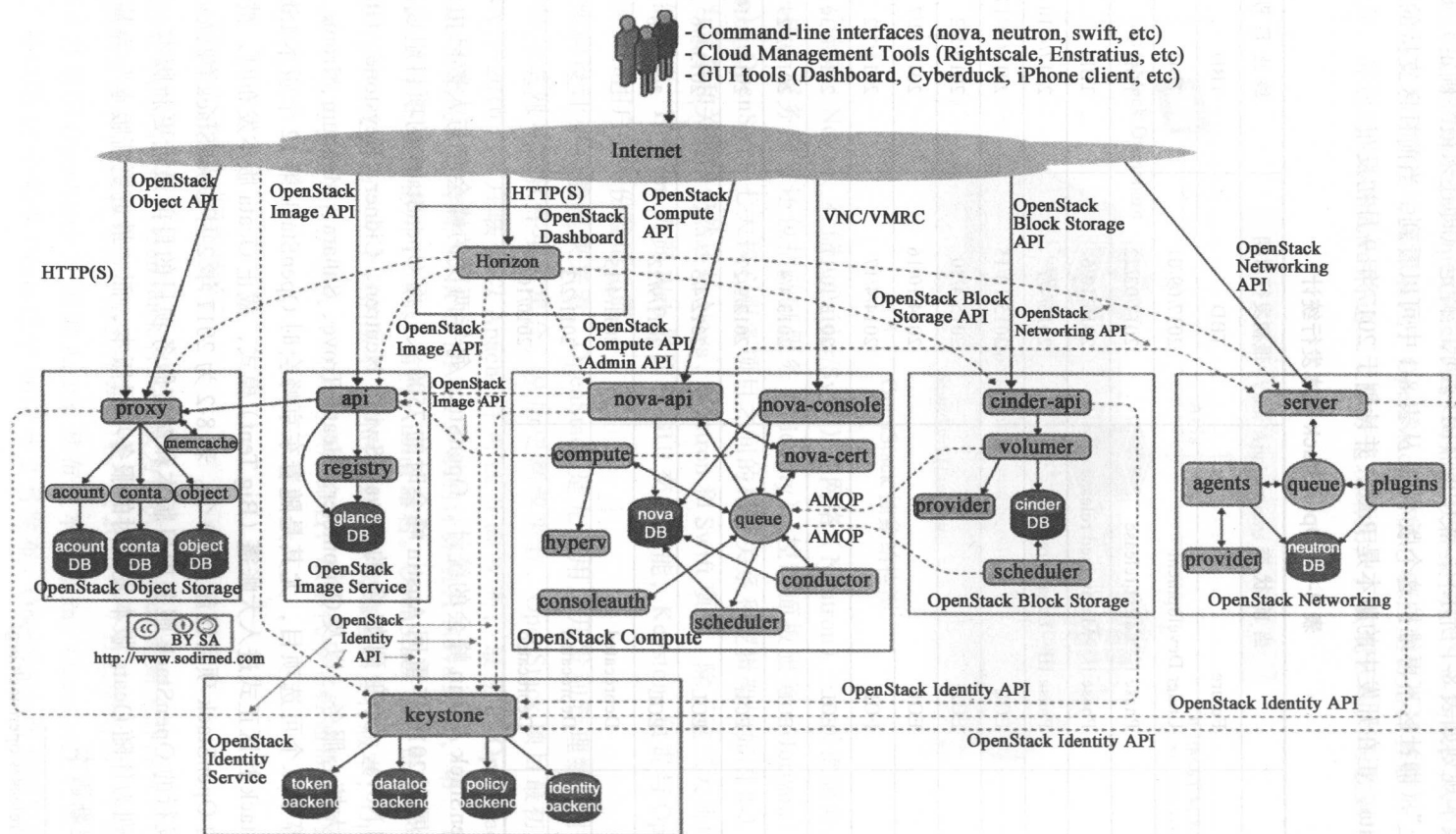


图 8-2 OpenStack 逻辑架构

表 8-2 Ocata 中各个服务项目的版本发行情况

服务 项 目	最 早 版 本	最近更新版本	服 务 项 目	最 早 版 本	最近更新版本
aodh	4.0.0	4.0.0	monasca-log-api	1.2.0	1.4.0
barbican	4.0.0	4.0.0	murano	3.1.0	3.2.0
ceilometer	8.0.0	8.0.0	neutron	10.0.0	10.0.0
cinder	10.0.0	10.0.0	nova	15.0.0	15.0.0
cloudkitty	5.0.0	5.0.0	panko	2.0.0	2.0.1
congress	5.0.0	5.0.0	sahara	6.0.0	6.0.0
designate	4.0.0	4.0.0	searchlight	2.0.0	2.0.0
freezer	4.0.0	4.0.0	senlin	3.0.0	3.0.0
glance	14.0.0	14.0.0	solum	5.1.0	5.2.0
heat	8.0.0	8.0.0	swift	2.11.0	2.13.0
horizon	11.0.0	11.0.0	tacker	0.6.0	0.7.0
ironic	7.0.0	7.0.0	tricircle	3.0.0	3.0.0
keystone	11.0.0	11.0.0	trove	7.0.0	7.0.0
magnum	4.0.0	4.1.0	vitrage	1.4.0	1.5.1
manila	4.0.0	4.0.0	watcher	0.32.0	1.0.1
mistral	4.0.0	4.0.0	zaqar	4.0.0	4.0.0
monasca-api	1.4.0	1.6.0			

在 Liberty 版本之前, OpenStack 的源代码版本发行号与 Liberty 之后的版本号完全不一样。由于 OpenStack 每年发行两个版本, 因此 Liberty 之前的服务项目版本号是以年份命名的。如 Icehouse 为 2014 年的第一个版本, 因此对应 Icehouse 的服务项目版本全部为 2014.1 版本, 而 Juno 版本为 2014 年的第二个版本, 因此对应 Juno 的服务项目版本全部为 2014.2 版本。即在 Liberty 之前, OpenStack 各个服务项目版本之间并不会像 Liberty 或 Ocata 一样遵循自己的差异化版本发行, 而是统一按照年份日期来统一命名, 表 8-3 即是以年份命名的 Juno 版本中各个服务项目的版本号。

表 8-3 Juno 版本中各个服务项目版本发行情况

服 务 项 目	最 早 版 本	最近更新版本	服 务 项 目	最 早 版 本	最近更新版本
ceilometer	2014.2	2014.2.4	keystone	2014.2	2014.2.4
cinder	2014.2	2014.2.4	neutron	2014.2	2014.2.4
glance	2014.2	2014.2.4	nova	2014.2	2014.2.4
heat	2014.2	2014.2.4	sahara	2014.2	2014.2.4
horizon	2014.2	2014.2.4	trove	2014.2	2014.2.4

8.1.3 OpenStack 组织机构

2010年7月,托管服务供应商 Rackspace 和美国国家航空航天局(NASA)合作,分别贡献出 Rackspace 云文件平台代码(Swift)和 NASA Nebula 平台代码(Nova),以 Apache 许可证授权将其开源发布,并将项目命名为 OpenStack,这便是 OpenStack 项目的诞生背景。在 OpenStack 开源项目成立之初,仅包含了 Swift 和 Nova 两个服务项目,并且社区主要以 Rackspace 为领导(Rackspace 也是社区最大的贡献和推动者)。因此,在 OpenStack 项目启动的前两年,Rackspace 监管着 OpenStack 社区的发展。但在 2011 年,OpenStack 项目领导人认为以供应商中立的方式才能更好地管理 OpenStack 开源项目,就像 Linux 基金会管理 Linux 操作系统内核一样。关于 OpenStack 基金会的作用,OpenStack 基金会执行董事 Jonathan Bryce 的解释是:“将 OpenStack 交给独立于供应商的中立基金会来管理,才能确保不出现供应商锁定的情况”。

2012 年 9 月,OpenStack 社区将 Nova 项目中的网络模块和块存储模块剥离出来,成立了两个新的核心项目,分别是 Quantum(即 Neutron 的前身)项目和 Cinder 项目,并发行了 OpenStack 的第六个版本 Folsom。与此同时,非盈利性组织 OpenStack 基金会宣布成立,该基金会主席由 SUSE 的行业计划、新兴标准和开源部门总监兼 Linux 基金会董事 Alan Clark 担任。成立之初,基金会拥有 24 名成员,并获得了 1000 万美元的赞助基金,Rackspace 的 Jonathan Bryce 担任常务董事。从此,OpenStack 社区计划今后 OpenStack 项目都由 OpenStack 基金会管理,同时 Rackspace 向基金会转交了社区管理活动和 OpenStack 商标。OpenStack 基金会设有三个分支,分别是董事会、技术委员会和用户委员会。

(1) 董事会

董事会负责为 OpenStack 基金会提供战略和财务监督,由 24 名成员组成,其中 8 名成员来自白金赞助商,8 名成员来自黄金赞助商,还有 8 名成员来自个人独立董事。董事会成员都是分散的,因此不会有任何独立实体对董事会具有过大的影响力。由于 OpenStack 基金会只设有 8 个白金赞助商,所以每个白金赞助商在董事会仅拥有一席之地。目前 OpenStack 基金会白金会员企业如图 8-3 所示,这些企业分别是 AT&T、Canonical、Hewlett Packard Enterprise(惠普企业)、IBM、Intel、Rackspace、Red Hat 和 SUSE。此外,虽然基金会目前有 19 家黄金会员企业(如图 8-4 所示),但是也只有 8 个董事会席位,这 8 个董事席位由 19 家黄金会员企业在一天内投票选出,投票过程不对社区开放。在董事会的成员构成中,没有任何一家公司可以在董事会拥有超过两个席位,即使董事会成员中的一家公司收购了董事会中的另一家公司。在董事会投票决定问题时,每个成员都只有一票,而当董事会成员投票法无法解决某个问题时,董事会主席可以投多票。

在董事会的选举中,白金会员的 8 个席位已基本固定,即由 8 个白金会员企业自己内部推举相关人士来兼任董事会成员,而黄金会员的 8 个席位则从目前的 19 家黄金会员企业中诞生,选举范围已基本固定。因此,对于未在上述 27 家白金和黄金会员企业任职的

个人，要想成为 OpenStack 基金会董事，唯一的方式就是通过社区选举成为个人独立董事，然而名额也非常有限，个人董事会成员的选举需要在一周之内从社区上万名会员中选举产生，而且仅有 8 个名额。

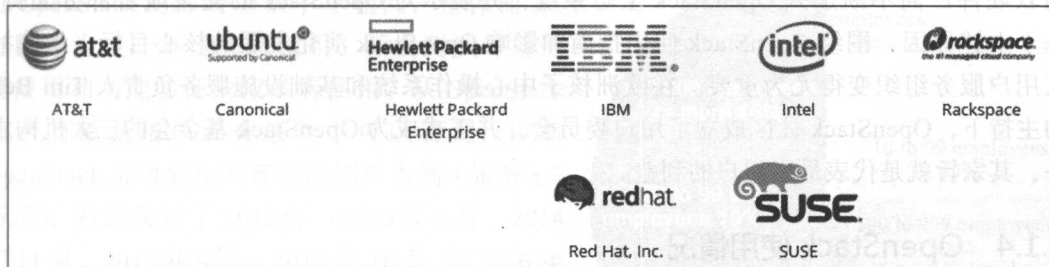


图 8-3 OpenStack 基金会白金会员

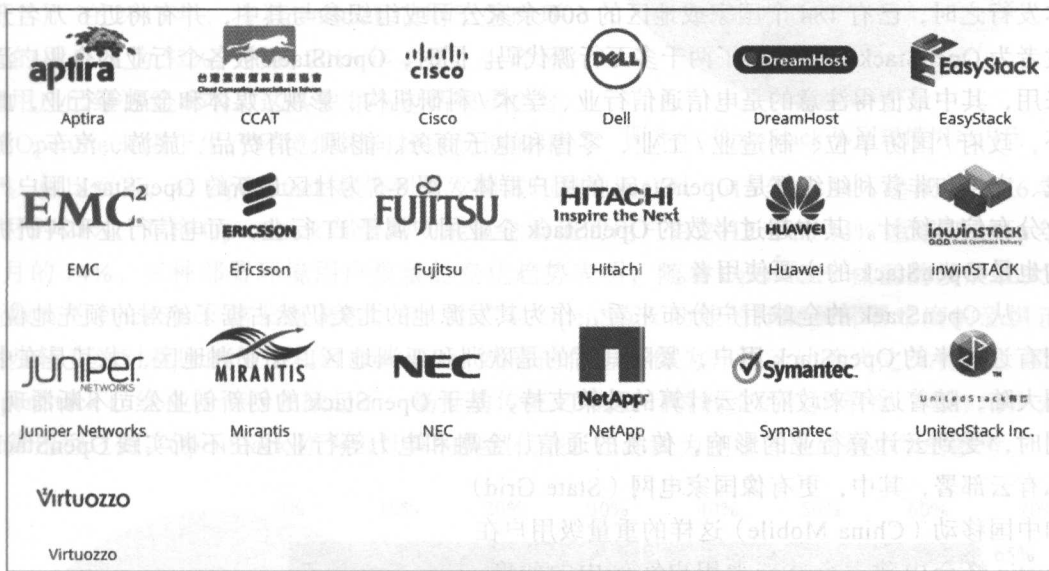


图 8-4 OpenStack 基金会黄金会员

(2) 技术委员会

技术委员会 (Technical Committee, TC) 是之前 OpenStack 的项目政策委员会 (Project Policy Board, PPB) 的延续，TC 的主要任务是负责 OpenStack 整体项目的技术领导和监督，规范 OpenStack 的技术发展方向，例如项目的开放性、透明性、通用性、集成和质量等，同时 TC 还要对影响到多个 OpenStack 服务项目的关键问题做出决定，并组成关于 OpenStack 项目技术决定的最终上诉委员会。总体而言，TC 通常就是负责对 OpenStack 的技术发展进行全程监督和决策。

(3) 用户委员会

在 2012 年 OpenStack 基金会成立之初，基金会内部仅有两个初始机构，即管理委员会

和技术委员会，管理委员会负责基金会战略制定以及资源使用和人员分配的监督，而技术委员会负责 OpenStack 社区的技术发展方向，后来管理委员会便演变为现在的董事会。但是随着 OpenStack 的发展，越来越多的终端用户将 OpenStack 部署于生产环境，并且大量的软硬件厂商不断加入 OpenStack 生态系统，并表示对 OpenStack 的完全或者部分支持。基于上述原因，围绕 OpenStack 使用指南和影响 OpenStack 演化发展的核心目标，组建社区用户服务组织变得尤为重要。在欧洲核子中心操作系统和基础设施服务负责人 Tim Bell 的主持下，OpenStack 社区成立了用户委员会，并正式成为 OpenStack 基金会的三大机构之一，其宗旨就是代表最终用户的利益。

8.1.4 OpenStack 使用情况

OpenStack 项目聚集了来自全球各地的云计算专家、开发者和运维人员，截至 Ocata 版本发行之时，已有 184 个国家或地区的 600 余家公司或组织参与其中，并有将近 6 万名开发者为 OpenStack 项目贡献了两千多万行源代码。同时，OpenStack 被各个行业的企业广泛采用，其中最值得注意的是电信通信行业、学术 / 科研机构、影视 / 媒体和金融等行业，此外，政府 / 国防单位、制造业 / 工业、零售和电子商务、能源、消费品、旅游、汽车、游戏、广告和非营利组织都是 OpenStack 的用户群体。图 8-5 为社区最新的 OpenStack 用户行业分布信息统计，其中超过半数的 OpenStack 企业用户属于 IT 行业，而电信行业和科研机构也是 OpenStack 的主要使用者。

从 OpenStack 的全球用户分布来看，作为其发源地的北美仍然占据了绝对的领先地位，拥有近过半的 OpenStack 用户，紧随其后的是欧洲和亚洲地区。在亚洲地区，尤其是在中国大陆，随着近年来政府对云计算的政策支持，基于 OpenStack 的创新创业公司不断涌现。同时，受到云计算行业的影响，传统的通信、金融和电力等行业也在不断实践 OpenStack 私有云部署，其中，更有像国家电网（State Grid）和中国移动（China Mobile）这样的重量级用户在 Austin 峰会中独占全球五大用户案例中的两席，而中国移动更是成为了 Newton 峰会 OpenStack 超级用户的获得者。

众多的用户实践表明，OpenStack 作为一个高弹性的开源云平台，完全具备支撑大规模企业云计算资源的需求，而在最新的 OpenStack 使用者用户企业规模调查报告中，超过 10 万员工规模的企业用户已经占据了相当比例。如图 8-6 所示，OpenStack 在不同规模企业中的用户占比相对平均，这也充分说明 OpenStack 作为弹性开源云计算平台，适合不同规模规模的用户使用，不论是

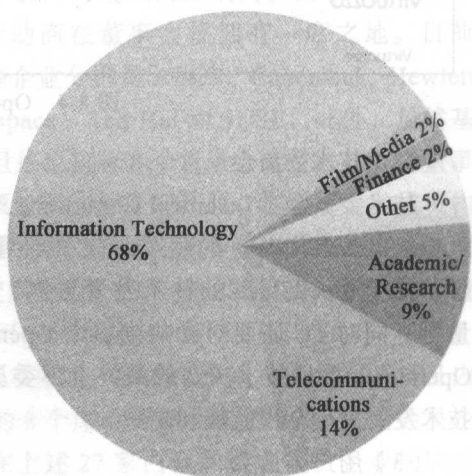


图 8-5 OpenStack 用户行业分布

小规模初创企业还是具有一定规模的大中型企业，OpenStack 都能够为用户提供适合企业自身的开源云计算能力。

在部署 OpenStack 的用户群中，不同的用户对 OpenStack 的部署场景不尽相同，整体而言，可以分为生产环境部署使用、开发测试环境部署使用和功能验证阶段部署使用。从 OpenStack 社区的最新用户调查报告中可以看到，以将 OpenStack 部署到生产环境的用户为例（如图 8-7 所示），社区统计了 2013 年、2014 年 5 月、2014 年 11 月、2015 年 4 月、2015 年 10 月和 2016 年 4 月 OpenStack 的部署使用情况（时间顺序由下到上）。在 2013 年，将 OpenStack 应用于生产环境的用户仅为 32%，而到 2016 年 4 月，生产环境用户已经达到 65%，这其中并不包括很多已经将 OpenStack 用于生产环境但未向社区反馈的用户。

相比而言，将 OpenStack 部署到开发测试环境的用户从 2013 年的 34% 递减到 2016 年 4 月的 21%，而将 OpenStack 部署为功能概念验证的用户由 2013 年的 34% 递减到 2016 年 4 月的 14%。三种部署环境用户数量的变化趋势表明，随着 OpenStack 社区的发展壮大，新版本的 OpenStack 越来越成熟稳定，用户对 OpenStack 功能的怀疑和部署难度的恐惧正逐步转变为对社区的信任和对 OpenStack 部署的信心。作为开源云计算当之无愧的王者，OpenStack 已经在各个行业掀起了一场推动传统 IT 向云计算转变的技术革命，而在用户云计算选型之初，OpenStack 作为云计算的典型代表，已然成为企业首选或是必须的参考。

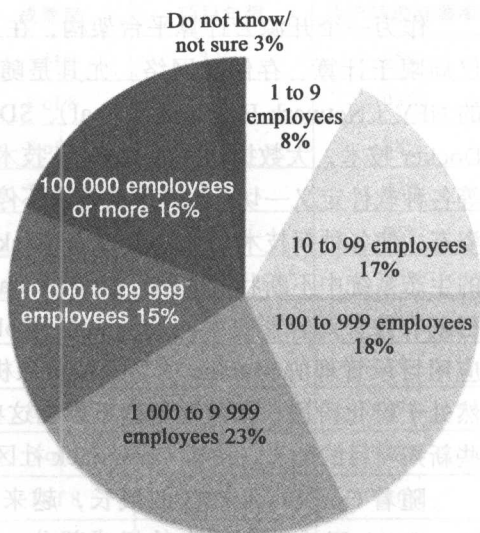


图 8-6 OpenStack 不同规模用户占比

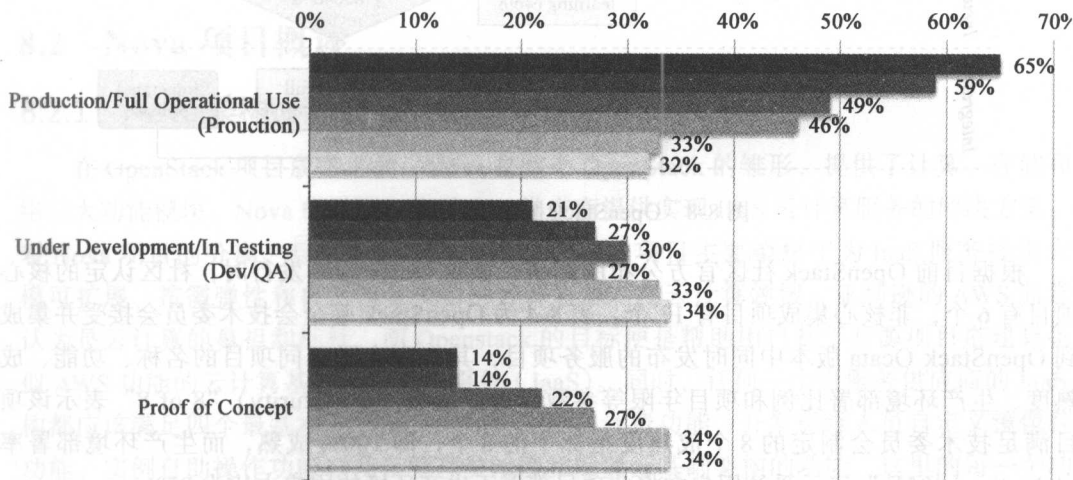


图 8-7 OpenStack 在用户不同部署环境中的变化

8.1.5 OpenStack 服务项目

作为一个开源云计算平台架构，在当前的大帐篷模式下，OpenStack 的功能已经不仅仅局限于计算、存储和网络。尤其是随着各个领域新技术的不断发展，面对如网络相关的 NFV (Network Function Virtual)、SDN (Software Define Network) 技术，容器相关的 Docker 技术，大数据相关的 Hadoop 技术，存储相关的 SDS (Software Define Storage) 技术等各种软件定义一切 (SDX) 的技术不停出现和发展，OpenStack 呈现出了极为开放包容的姿态，使各种新技术都可以在 OpenStack 的大帐篷下孵化成为独立项目，从而在 OpenStack 的生态系统中不断壮大成熟。截至 Ocata 版本，OpenStack 社区接受并不断壮大的新服务项目就有像针对容器技术的 Magnum、Kolla 项目，针对大数据的 Sahara、Trove 项目，针对应用目录管理的 Murano 项目，针对裸机部署的 Ironic、TripleO 等，尽管其中很多项目仍然处于孵化阶段，但是这丝毫不影响这些项目在 OpenStack 大帐篷中的成长，同时由于这些新兴项目的加入，使得 OpenStack 社区的影响力和用户数都达到了空前的规模。

随着 OpenStack 社区的成长，越来越多的新项目加入 OpenStack 社区，并期望成为 OpenStack 正式发布版本的组成部分，而任何一个想要成为 OpenStack 官方认可并集成发布的新项目，都需要经过 OpenStack 技术委员会规定的项目成长三阶段，即外部独立开发 (External) 阶段、OpenStack 项目孵化 (Incubated) 阶段和 OpenStack 官方集成发布 (Integrated) 阶段，具体流程如图 8-8 所示。

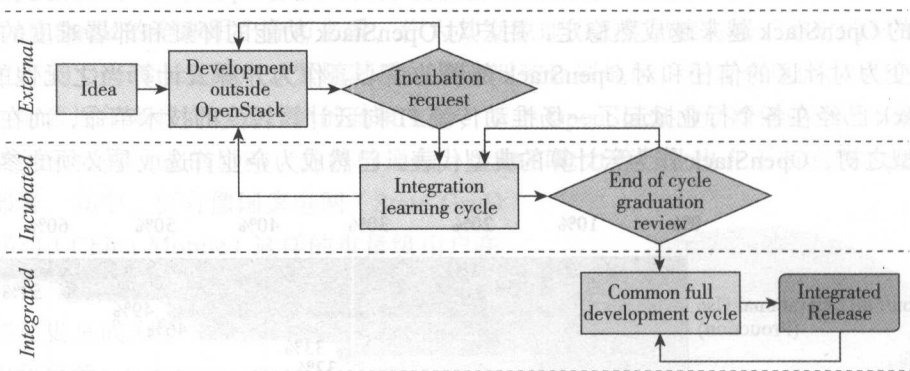


图 8-8 OpenStack 服务项目发展三阶段

根据目前 OpenStack 社区官方公布的结果，截止 Ocata 版本发行时，社区认定的核心项目有 6 个，非核心集成项目有 13 个。表 8-4 为 OpenStack 基金会技术委员会接受并集成到 OpenStack Ocata 版本中同时发布的服务项目，其中列出了不同项目的名称、功能、成熟度、生产环境部署比例和项目年限等参数，这里成熟度 (Maturity) “8 of 8” 表示该项目满足技术委员会制定的 8 个成熟度指标中的 8 个，即 100% 成熟，而生产环境部署率 (Adoption) “95%” 表示受访用户中将此项目部署于生产环境的用户占比为 95%。

表 8-4 OpenStack Ocata 版本集成发布项目列表

项目代号	服务名称	是否核心项目	成熟度	项目年限	生产环境部署率
Nova	Compute	Y	8 of 8	7 Yrs	95%
Neutron	Networking	Y	8 of 8	5 Yrs	93%
Cinder	Block Storage	Y	8 of 8	5 Yrs	88%
Swift	Object Storage	Y	7 of 8	7 Yrs	52%
Keystone	Identity	Y	7 of 8	5 Yrs	96%
Glance	Image Service	Y	6 of 8	7 Yrs	95%
Horizon	Dashboard	N	6 of 8	5 Yrs	87%
Ceilometer	Telemetry	N	1 of 8	4 Yrs	55%
Heat	Orchestration	N	6 of 8	4 Yrs	67%
Trove	Database	N	3 of 8	3 Yrs	13%
Sahara	Elastic Map Reduce	N	3 of 8	3 Yrs	10%
Ironic	Bare-Metal Provisioning	N	5 of 8	3 Yrs	21%
Zaqar	Messaging Service	N	4 of 8	3 Yrs	4%
Manila	Shared Filesystems	N	5 of 8	3 Yrs	14%
Designate	DNS Service	N	3 of 8	3 Yrs	16%
Barbican	Key Management	N	4 of 8	3 Yrs	9%
Magnum	Containers	N	2 of 8	2 Yrs	11%
Murano	Application Catalog	N	1 of 8	2 Yrs	11%
Congress	Governance	N	1 of 8	2 Yrs	2%

8.2 Nova 项目概述

8.2.1 Nova 架构设计

在 OpenStack 项目成立之初，Nova 是整个 OpenStack 的雏形，提供了计算、存储和网络三大功能模块。Nova 的设计初衷是为云供应商提供实现 IaaS 云计算服务的解决方案。随着 Nova 项目的分离，目前 Openstack 社区的 Nova 项目主要聚焦于为 IaaS 服务提供大规模可扩展、按需弹性和自助服务供给的计算资源。在云计算领域，亚马逊的 AWS 通常被认为是云计算的鼻祖和标杆，而 Openstack 的目标便是帮助用户基于开源项目搭建具备类似 AWS 功能的云计算基础架构设施服务（IaaS）。同时，任何一个云服务供应商的 IaaS 架构都应该满足四个最基本的功能，即用户注册与计费功能、开发运维人员自定义镜像存储功能、实例自助操作功能以及云管理员治理和配置云基础架构的功能，这里的每一个功能都可以通过不同的模块来实现。一个简单的云服务供应商 IaaS 概念设计架构如图 8-9 所示，

图中将 IaaS 架构分为资源层、逻辑层和呈现层，并集成了计费、认证和监控等功能模块。其中，资源层包括了计算、存储和网络三大云计算核心资源，主要功能是为用户提供计算服务、存储服务和网络服务；逻辑层也称为控制层，主要是为整个云计算平台提供策略依据、调度过滤算法、应用及实例编排、日志记录以及智能自动化运维等功能；呈现层位于云计算平台的最外层（也称为展示层），主要功能是将云计算资源以用户便于接受和操作的方式呈现给最终的云平台用户；而根据最终用户的角色，又可以将云平台用户划分为应用开发人员、应用运维人员、应用使用者和云平台管理人员，不同的用户群将通过呈现层所提供的不同 API 进行资源调用和管理。

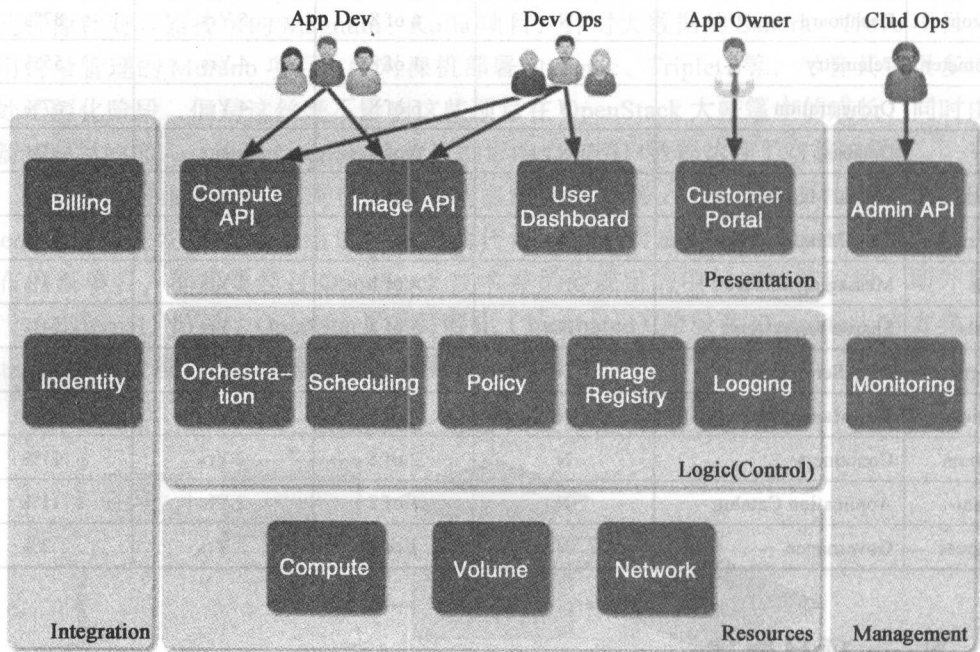


图 8-9 云服务供应商 IaaS 概念架构

为了满足图 8-9 的云计算平台概念设计架构，OpenStack 相应构建了不同的功能模块，而每个功能模块都对应着彼此独立的开源服务项目。从目前 OpenStack 社区集成发布的服务项目来看，OpenStack 集成发布的项目以及正在孵化中的项目几乎完全覆盖了图中的功能概念模块。这些服务项目到概念模块的映射如图 8-10 所示，其中的 Nova 项目对资源层、逻辑层和呈现层中相应的概念功能都有所覆盖，即 Nova 不仅实现了资源层中的计算功能，还实现了逻辑层和呈现层中的相关功能。因此，尽管 OpenStack 的 F 版本将 Nova-volume 和 Nova-network 进行了分离，但是 Nova 对于 OpenStack 云平台的重要性仍然是不言而喻的，也可以认为 Nova 是 OpenStack 中最为核心和成熟的项目。

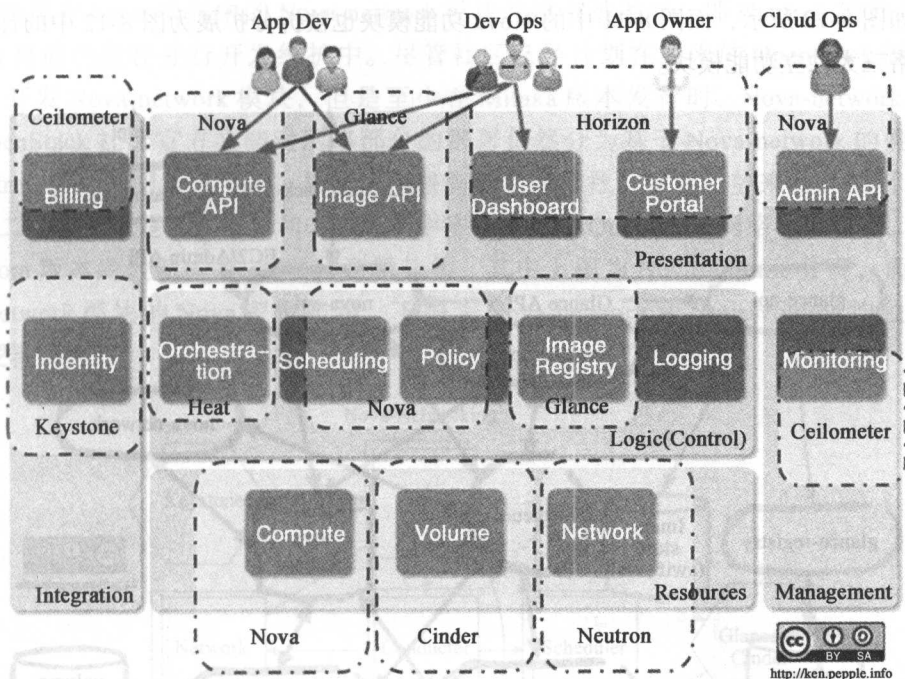


图 8-10 Nova 在 IaaS 概念架构中的功能映射

Nova 项目最初的源代码由美国国家航空航天局 (NASA) 贡献, 截至 Ocata 版本, Nova 项目已发行了 15 个版本, 也是社区所有项目中最成熟和用户生产环境部署率最高的项目。在 2010 年 OpenStack 项目成立之初, Nova 项目主要分为 Nova-Compute、Nova-volume 和 Nova-network 三大功能模块。在 2012 年 9 月 OpenStack 的 Folsom 版本发行时, 社区才将 Nova-volume 和 Nova-network 独立出来分别构建了 Cinder 和 Quantum 项目 (后因商标原因更名为 Neutron 项目)。在 OpenStack 的 A 至 E 版本中, OpenStack Nova 项目的逻辑架构如图 8-11 所示, 其中, 除了 Nova-Compute、Nova-volume 和 Nova-network 三大功能模块之外, 还有处理 RESTful API 请求的 Nova-API 模块、调度 Nova-Compute 的 Nova-scheduler 模块、用以模块信息交互的消息队列系统和配置及状态数据存储的数据库。而在早期的 OpenStack 版本中, 仅有 Nova、Swift 和 Glance 三大项目, 如果用户不准备使用对象存储 Swift, 则 Nova 和 Glance 项目即构成了早期的 OpenStack 云平台。

在 OpenStack 的 Folsom 版本发行后, Nova-volume 和 Nova-network 被独立成为块存储 Cinder 项目和网络 Quantum (Neutron) 项目, 而 Nova 自身的功能模块也被不断细分, 除了 Nova-Compute 和 Nova-API 功能模块, 以及消息队列和数据库之外, Nova 项目还构建了 Nova-cert、Nova-Conductor、Nova-consoleauth 和 nova-console 等模块。块存储 Cinder 项目和网络服务 Neutron 独立后, OpenStack 中三大核心功能计算、存储和网络项目之间的逻辑

辑架构如图 8-12 所示，而图 8-11 中的 Nova 功能模块也被拆分扩展为图 8-12 中的计算、存储和网络三大独立功能模块。

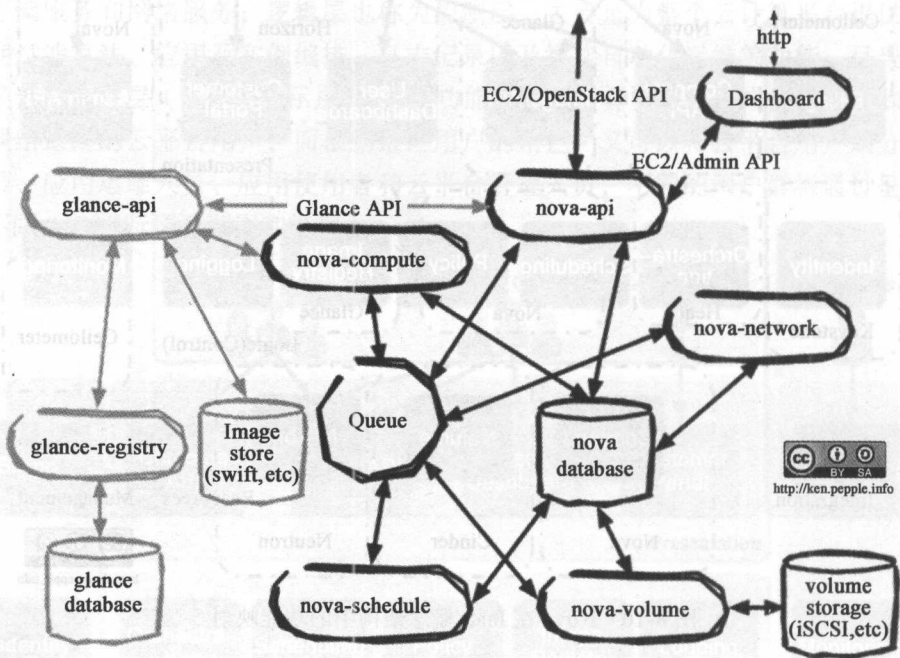


图 8-11 Folsom 版本之前的 Nova 逻辑架构

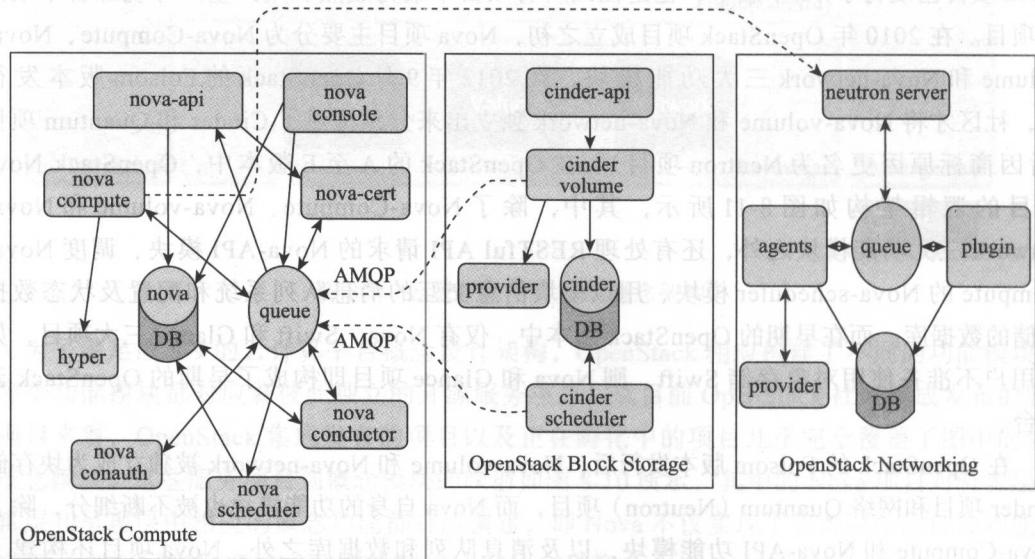


图 8-12 Nova、Cinder 和 Neutron 逻辑架构

值得注意的是，虽然 OpenStack 的 Nova-network 模块已经独立成为目前的 Neutron 项

目,但是原有 Nova 项目中的 Nova-network 模块并没有被抛弃,即 Neutron 项目与 Nova-network 目前仍然在并行开发维护中。尽管社区已经计划在未来的 OpenStack 版本中不再维护开发 Nova-network 模块,但是至少在 Mitaka 版本发行时,Nova-network 仍然可用,OpenStack 社区官方文档对网络部分的部署仍然分为基于 Nova-network 的网络和基于 Neutron 项目的网络。当然,从使用网络高级功能和社区发展趋势来看,使用 Neutron 将是不二选择,也是社区推荐和未来版本中唯一支持的 OpenStack 网络功能项目。图 8-13 为 Folsom 版本后最新的 Nova 项目逻辑架构,其中上图为 Nova-volume 分离,而包含有 Nova-network 模块的 Nova 逻辑架构,下图为 Nova-network 和 Nova-volume 均独立后的 Nova 逻辑架构。

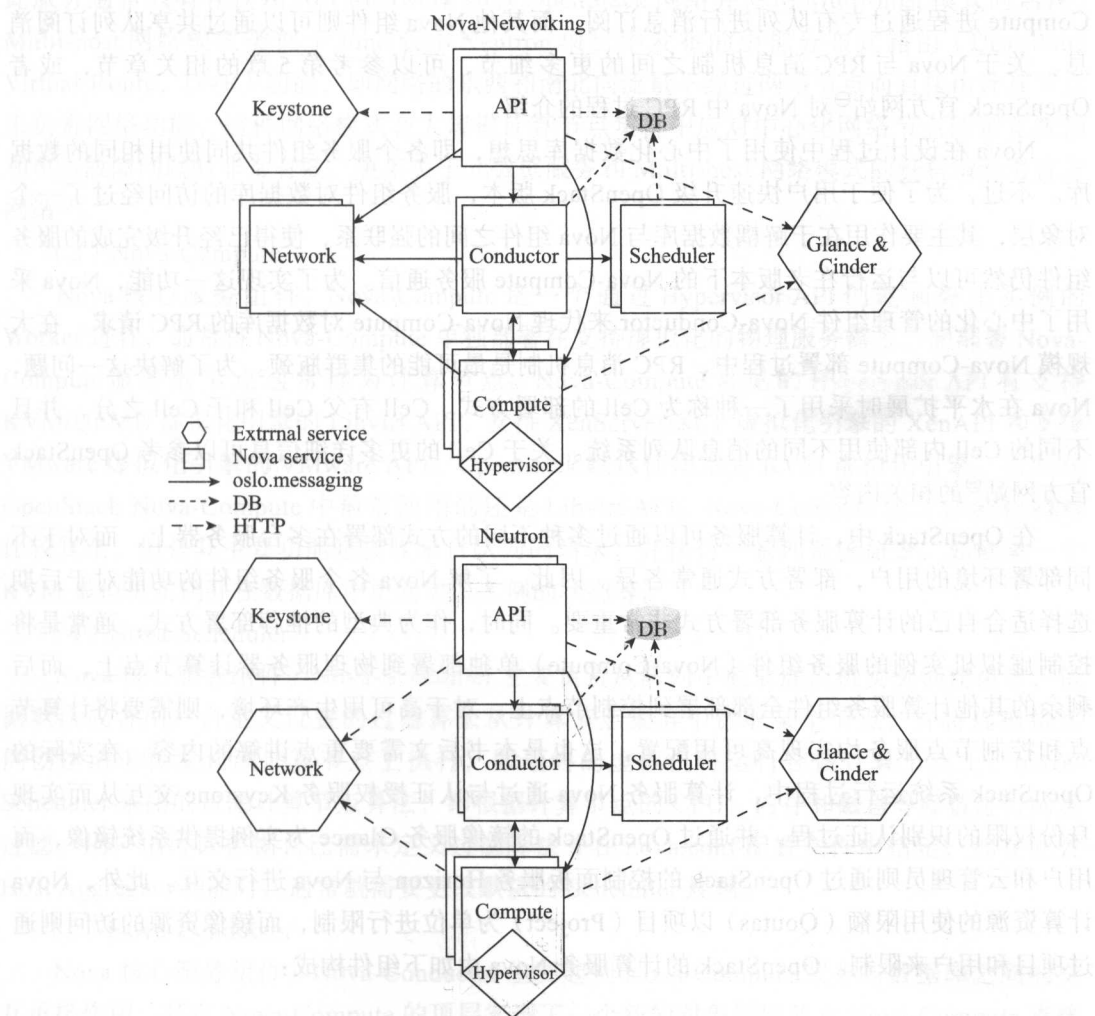


图 8-13 Folsom 版本之后的 Nova 逻辑架构

8.2.2 Nova 功能模块

Nova 由负责不同功能的服务进程所构成，其对外提供的服务接口为 REST API，而各个内部组件之间通过 RPC 消息传递机制进行通信。Nova 中提供 API 请求处理功能的模块是 Nova-API，由 API 服务进程来处理数据库读写请求、向其他服务组件发送 RPC 消息的请求和生成 RPC 调用应答的请求等。在 Nova 中，RPC 消息机制通过 oslo.messaging 库实现，这里的 oslo.messaging 库是对消息队列系统的顶层抽象。Nova 中的大部分服务组件，除了 Nova-Compute，都能以分布式的方式运行在多台服务器上，并且各服务组件之间会使用一个 Manager 进程来监听 RPC 消息。Nova-Compute 组件的特别之处在于其作为一个独立进程运行在某个 Nova-Compute 管理下的 Hypervisor 上，从 RPC 消息机制而言，Nova-Compute 进程通过专有队列进行消息订阅，而其他 Nova 组件则可以通过共享队列订阅消息。关于 Nova 与 RPC 消息机制之间的更多细节，可以参考第 5 章的相关章节，或者 OpenStack 官方网站^①对 Nova 中 RPC 过程的介绍。

Nova 在设计过程中使用了中心化数据库思想，即各个服务组件共同使用相同的数据库。不过，为了便于用户快速升级 OpenStack 版本，服务组件对数据库的访问经过了一个对象层，其主要作用在于解耦数据库与 Nova 组件之间的强联系，使得已经升级完成的服务组件仍然可以与运行在老版本下的 Nova-Compute 服务通信。为了实现这一功能，Nova 采用了中心化的管理组件 Nova-Conductor 来代理 Nova-Compute 对数据库的 RPC 请求。在大规模 Nova-Compute 部署过程中，RPC 消息机制是最可能的集群瓶颈。为了解决这一问题，Nova 在水平扩展时采用了一种称为 Cell 的部署方式。Cell 有父 Cell 和子 Cell 之分，并且不同的 Cell 内部使用不同的消息队列系统。关于 Cell 的更多详细信息可以参考 OpenStack 官方网站^②的相关内容。

在 OpenStack 中，计算服务可以通过多种不同的方式部署在多台服务器上，而对于不同部署环境的用户，部署方式通常各异。因此，了解 Nova 各个服务组件的功能对于后期选择适合自己的计算服务部署方式非常重要。同时，作为典型的推荐部署方式，通常是将控制虚拟机实例的服务组件（Nova-Compute）单独部署到物理服务器计算节点上，而后剩余的其他计算服务组件全部部署到控制节点上。对于高可用生产环境，则需要将计算节点和控制节点服务均实现高可用配置，这也是本书后文需要重点讲解的内容。在实际的 OpenStack 系统运行过程中，计算服务 Nova 通过与认证授权服务 Keystone 交互从而实现身份权限的识别认证过程，并通过 OpenStack 的镜像服务 Glance 为实例提供系统镜像，而用户和云管理员则通过 OpenStack 的控制面板服务 Horizon 与 Nova 进行交互。此外，Nova 计算资源的使用限额（Quotas）以项目（Project）为单位进行限制，而镜像资源的访问则通过项目和用户来限制。OpenStack 的计算服务 Nova 由如下组件构成：

① <http://docs.openstack.org/developer/nova/rpc.html>

② <http://docs.openstack.org/developer/nova/cells.html>

(1) Nova-API

Nova API 服务组件。Nova-API 服务负责接收和响应终端用户对 OpenStack 计算资源发起的 API 调用请求，如 WSGI APP 路由请求和授权相关请求。Nova-API 接收到请求后，通常将请求转发给 Nova 的其他组件进行处理，如 Nova-scheduler。Nova-API 除了支持 OpenStack Compute API，还支持 AWS EC2 API 和授权用户执行管理任务的特定 API。Nova-API 遵循特定的策略并初始化大部分的编排操作，如用户发起一个创建实例的请求，则创建实例的初始编排工作由 Nova-API 首先发起。

(2) Nova-API-Metadata

Nova API 服务组件。Nova-API-Metadata 服务主要用于接收来自实例的元数据请求，此服务通常只有在使用 Nova-network 部署 OpenStack 网络并使用 Multi-host 模式时启用。Multi-host 网络模式类似于 Juno 版本 Neutron 项目中发布的虚拟分布式路由（Distribute Virtual Route, DVR）功能，即网络的东西和南北向流量不经过网络节点而直接由计算节点来负责网络功能，这种网络模式在大规模计算节点集群中应对中心化网络节点的带宽瓶颈和单点故障问题时非常有效。更多关于元数据服务和 Multi-host 网络模式的资料请参考官方网站^①。

(3) Nova-Compute

Nova 核心服务组件。Nova-Compute 是一个通过 Hypervisor API 创建和终止实例的 Worker 进程，通常将 Nova-Compute 单独部署在支持虚拟化的物理服务器上，而部署 Nova-Compute 服务的节点通常称为计算节点。Nova-Compute 常见的 Hypervisor API 有支持 KVM/QEMU 虚拟化引擎的 Libvirt API、支持 XenServer/XCP 虚拟化引擎的 XenAPI 和支持 VMware 虚拟化引擎的 VMware API。OpenStack 默认使用的是 KVM 虚拟化引擎，因此在 OpenStack Nova-Compute 中最常使用的还是 Libvirt API。Nova-Compute 的工作流程相对比较复杂，但是其主要功能是接收来自队列的请求，并执行一系列系统命令，如创建一个 KVM 虚拟机实例并在数据库中更新对应实例的状态等。

(4) Nova-Scheduler

Nova 核心服务组件。Nova-Scheduler 主要负责从队列中截取虚拟机实例创建请求，依据默认或者用户自定义设置的过滤算法从计算节点集群中选取某个节点，并将虚拟机实例创建请求转发到该计算节点上执行，即最终的虚拟机将运行在该计算节点上。Nova-Scheduler 采用的过滤计算节点算法，即根据计算节点的 CPU、内存和磁盘等参数进行筛选过滤。用户也可以根据自己需求定义过滤算法并在 nova.conf 配置文件中指定，如在配置 Host Aggregate 功能时，通常就需要更改默认的 Scheduler 规则。

(5) Nova-Conductor

Nova 核心服务组件。Nova-Conductor 主要起到 Nova-Compute 服务与数据库之间的交互承接作用，其在 Nova-Compute 的顶层实现了一个新的对象层以防止 Nova-Compute 直接

① <http://docs.openstack.org/admin-guide/compute-networking-nova.html#metadata-service>

访问数据库带来的安全风险。在实际运行中，Nova-Compute 并不直接读写访问数据库，而是通过 Nova-Conductor 实现数据库访问。Nova-Conductor 组件可以水平扩展到多个节点上同时运行，但是 Nova-Conductor 不能部署到运行 Nova-Compute 的计算节点上，否则将不能隔离 Nova-Compute 对数据库的直接访问，从而不能真正起到降低数据安全风险的作用。更多关于 Nova-Conductor 的详细信息请参考 OpenStack 的官方配置参考网站^①。

(6) Nova-Cert

Nova 核心服务组件。Nova-Cert 是服务器守候进程，主要为基于 X509 认证的 Nova Cert 提供服务，过去通常用于对 euca-bundle-image 镜像生成 X509 证书，目前仅在使用 EC2 API 时候才会启用。

(7) Nova-Network

虚拟机网络服务。Nova-Network 的功能类似 Nova-Compute，主要负责从队列中接收客户端的网络相关任务请求，并对任务请求进行相应的网络操作，例如执行网桥创建或者更改 IPTables 表规则等操作。在最新发行的几个 OpenStack 版本中，Nova-Network 的功能几乎没有太多变化，而对应的 Neutron 项目却是日新月异地不断完善各种高级功能（Neutron 项目可以认为是软件定义网络（SDN）的具体实现）。可见，作为比较古老的 Nova-Network，被社区淘汰也是早晚的事情，所以对于 OpenStack 的新用户，通常推荐直接使用 Neutron 来提供网络服务。

(8) Nova-Consoleauth

虚拟机控制台服务。Nova-Consoleauth 主要为虚拟机控制台连接提供认证授权服务。在运行 VNC 代理服务的 OpenStack 集群中，必须运行 Nova-Consoleauth 服务。一个 Nova-Consoleauth 实例可为多种类型的代理服务提供认证授权服务。需要注意的是，不要混淆 Nova-Consoleauth 与 Nova-Console，后者是 XenAPI 风格的控制台服务，而目前多数 VNC 代理软件都已经不再使用 Nova-Console。

(9) Nova-novncproxy

虚拟机控制台服务。Nova-novncproxy 主要提供对运行状态中的实例进行 VNC 连接访问的代理。其直接基于网页的 novnc 客户端连接，即支持基于 Web 网页的实例访问。这对于用户而言是个非常方便的功能：用户无须与虚拟机建立类似 SSH 的安全连接即可通过 Web 访问和操作虚拟机。

(10) Nova-xvncproxy

虚拟机控制台服务。Nova-xvncproxy 主要提供对运行状态中的实例进行 VNC 连接访问的代理，其仅支持特定于 OpenStack 的 Java 客户端发起的 VNC 连接。

(11) Nova-spicehtml5proxy

虚拟机控制台服务。Nova-spicehtml5proxy 主要提供对运行状态中的实例进行 SPICE 连接访问的代理，其支持基于浏览器的 HTML5 客户端虚拟机实例访问。SPICE 为 Redhat

① <http://docs.openstack.org/mitaka/config-reference/compute/conductor.html>

开源虚拟化桌面的主要组件之一，能够提供与物理桌面完全相同的最终用户体验，OpenStack 官方文档默认使用的虚拟机桌面访问方式为 NoVNC，如果用户需要开启 SPICE 协议，则需要将 NoVNC 关闭，具体配置过程可以参考相关网站^①。

(12) Nova Client

终端命令行工具。主要功能就是使用户可以向 OpenStack 的 Nova API 提交命令行，例如租户管理员或者终端用户。

(13) Database

Nova 之外的核心组件。主要用于存储云基础架构中对象的创建时和运行时状态，包括可用的实例类型、使用中的实例、可用的网络和项目等信息。理论上，OpenStack 支持任何类型的 SQL 数据库，但实际使用中，测试时用得最多的是 SQLite3 数据库，生产环境部署中用得最多的是 MySQL/MariaDB 或 PostgreSQL 数据库。更多 OpenStack 数据库的相关内容请参考第 7 章的相关内容。

(14) MessagesQueue

Nova 之外的核心组件。组件进程之间进行消息交换传递的中心 Hub，通常采用 RabbitMQ 实现，也可以通过 ZeroMQ 实现。更多消息队列系统的相关内容请参考本书第 5 章。

8.3 Nova 分区与区域

OpenStack 中的 Nova 被设计为具备大规模水平可扩展的功能，所以在 OpenStack 的架构设计之初，必须考虑到后期云计算资源的可扩展性。与亚马逊的 AWS 中将计算资源分为区域 (Region) 和可用域 (Availability Zone, AZ) 相同，OpenStack 也实现了 Region 和 AZ 的概念，同时还针对 Nova 的可扩展性设计了 Cell 和主机集合 (Host Aggregate) 的概念。其中，Region、AZ 和 Cell 都是对用户主观可见的区域划分，而 Host Aggregate 是对 Nova 计算节点的逻辑划分。这几个概念在 OpenStack 架构设计中的体现如图 8-14 所示，可以看到，不同 Region 之间是完全隔离的，而不同 Region 内部又可以划分为父 Cell 和子 Cell，Cell 内部可以再划分不同的 AZ，AZ 内部还可以再从逻辑上将不同类型的服务器划分为不同的 Host Aggregate。

从顶层架构设计来看，Region 概念范围大于 Cell，Cell 概念范围大于 AZ，AZ 概念范围大于 Host Aggregate，Region、Cell、AZ 和 Host Aggregate 之间的范畴关系如图 8-15 所示。

8.3.1 Nova 中的 Region

Region 是地理位置上隔离的数据中心区域，不同 Region 之间是彼此独立的，即某个 Region 范围内的人为或自然灾害并不会影响到其他 Region 的正常运行。Region 的概念通常在云服务供应商中较为常见，因为 Region 的多少是衡量一个公有云服务供应商运营成熟度

① <http://blog.csdn.net/tantexian/article/details/38728229>

的关键指标。对于云计算资源用户而言，要想实现自身业务系统的容灾和高可用设计，通常需要将相同业务系统部署到云服务供应商的不同 Region 中，同时借助负载均衡器才能实现容灾双活的功能。图 8-16 为 Region 概念的抽象拓扑，理想情况下，不同的 Region 最好是分别位于地球上不同的大洲之间，这种布局使得不同 Region 几乎不受某个地区大面积自然灾害的影响，当然也可以将不同的 Region 部署到同一国家的不同省份或州之间，具体如何布局 Region 更多是受成本预算导向的。

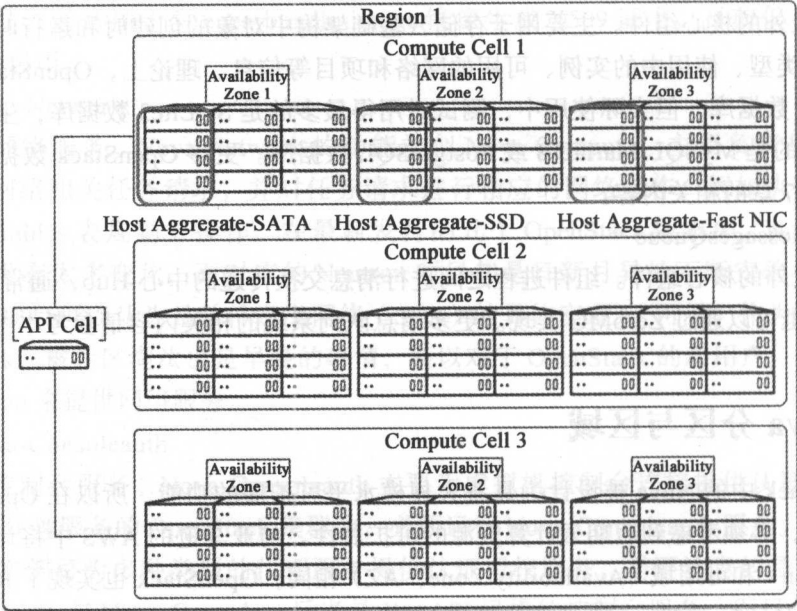


图 8-14 OpenStack 中的 Region、AZ、Cell 与 Host Aggregate

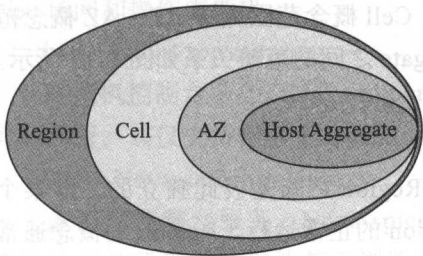


图 8-15 Region、AZ、Cell 与 Host Aggregate 概念范畴关系

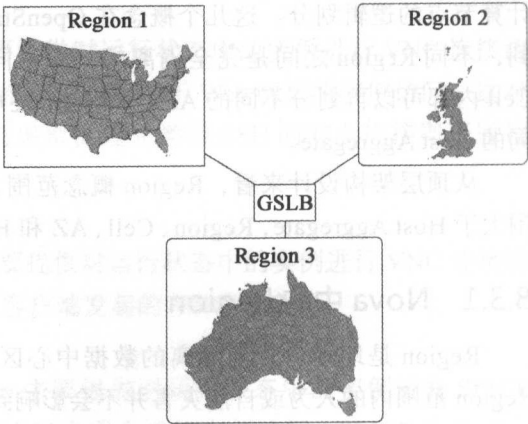


图 8-16 不同地理位置上的 Region

Region 的设计更多是为公有云服务供应商考虑的,对于私有云部署,如果不进行容灾高可用设计,则通常很少在架构之初考虑 Region。而对公有云而言,Region 通常是标配,不论是国外的云计算巨头 AWS、Softlayer、Azure,还是国内的阿里云、腾讯云、青云等,用户都可以在公有云服务商提供的不同 Region 中创建自己的服务器并部署业务系统,选择标准通常是以距离用户最近为首选。图 8-17 为青云服务门户所提供的 Region 选择,目前仅有广东 1 区、北京 2 区以及香港的亚太 1 区可供用户选择,如果用户位于广东或者深圳,则自然选择广东 1 区进行云服务创建。当然,如果有基于政策法规方面的考虑,由于大陆和香港的差异性,则可以考虑通过亚太 1 区进行云服务创建。

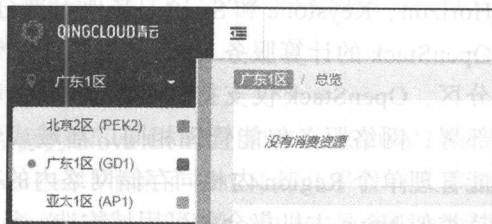


图 8-17 青云的 Region 实现

在具体的实现过程中,不同的 Region 通常共用相同的认证服务和控制面板服务。在 OpenStack 中,如果是基于 OpenStack 部署公有云,则多个 Region 之间通常共享同一个 Keystone 和 Dashboard 服务,而如果是基于 OpenStack 来部署私有云,并希望通过不同的 Region 来实现业务系统的高可用或者灾备,则通常还需要部署单一的共享存储池。不同的 Region 之间可以通过共享存储池进行数据复制同步来实现高可用(根据需求和实现技术,也可以部署隔离区域专有的存储池)。图 8-18 为 OpenStack 官方推荐的私有云多区域部署架构,Region1 和 Region2 共享 Horizon、Keystone 和 Swift 服务,其他 OpenStack 服务在 Region 中独立部署。由于不同 Region 内部服务独立部署,因而不同 Region 内部相同的 OpenStack 服务会有不同的 Endpoints API,对不同 Region 内部服务的访问就需要指定不同的 API,在实际应用中,用户通常是通过负载均衡器实现不同 Region 中服务 API 的访问。

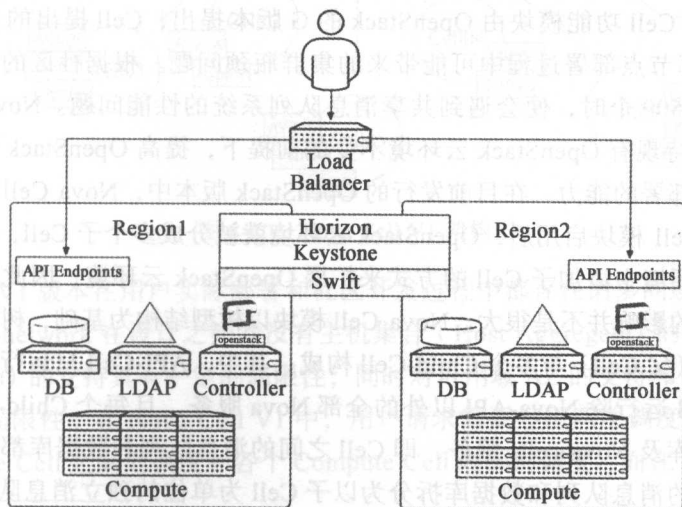


图 8-18 OpenStack 中 Region 架构设计

此外，在 OpenStack 的众多服务中，并非所有的服务都支持跨 Region 部署，除了 Horizon、Keystone 和 Swift，其他大部分服务都被设计为仅在同一个 Region 内运行，如 OpenStack 的计算服务 Nova 被设计用于管理单个 Region 内的计算资源。对于计算资源的分区，OpenStack 仅支持 Cell 部署、AZ 和 Host Aggregate 划分，并不能实现跨 Region 的部署；网络服务仅能管理相同广播域或者互联交换机集中的网络资源；块存储服务也只能管理单个 Region 内相同存储网络内的存储资源。在同一个 Region 内，块存储服务也支持类似 Nova 主机集合的可用域构造，从而将不同类型的存储划分到不同的可用域中。在 OpenStack 中配置多 Region 比较简单，以 Redhat 的 RDO 源为例。RDO 部署中默认的 Region 为 RegionOne，如果想要部署另外一套独立的 OpenStack，可以将其命名为 RegionTwo。由于 Keystone 要求不同 Region 使用相同的认证信息，简单起见，只需将 RegionOne 中的 Keystone 数据库表导出并在 RegionTwo 的控制节点上导入这些表到 Keystone 数据库，然后刷新 Dashboard 或者重新登录，则 OpenStack 的 Horizon 将会自动识别到新增的 RegionTwo，如图 8-19 所示。

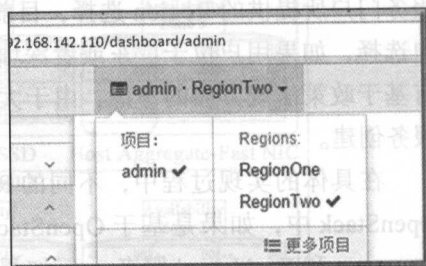


图 8-19 OpenStack 中多 Region 配置结果

从理论而言，Region 其实是个很灵活的设计概念。它的地理位置级别的计算资源隔离并不意味着只有相隔至少几十公里的数据中心之间才能部署，其理论范围可以很小，如位于相同数据中心同一个机柜或相邻机柜的服务器也可以被划分为不同的 Region，只不过这样的划分并无实际意义。

8.3.2 Nova 中的 Cell

Nova 组件的 Cell 功能模块由 OpenStack 的 G 版本提出，Cell 提出的主要需求是解决大规模 Nova 计算节点部署过程中可能带来的集群瓶颈问题。根据社区的讨论，通常在计算节点数目超过 500 个时，便会遇到共享消息队列系统的性能问题。Nova 的 Cell 功能模块允许用户在保持现有 OpenStack 云环境不变的前提下，提高 OpenStack 计算节点水平横向扩展和大规模部署的能力。在目前发行的 OpenStack 版本中，Nova Cell 功能默认并未启用，而当 Nova Cell 模块启用后，OpenStack 云环境就被分成多个子 Cell，并且是通过在原有 OpenStack 云环境中添加子 Cell 的方式来扩展 OpenStack 云环境，因此后期 Cell 功能的启用对原云环境的影响并不是很大。Nova Cell 模块以树型结构为基础，树形 Cell 架构主要由一个 API-Cell（父 Cell）与多个 Child-Cell 构成。其中，API-Cell 只运行 Nova-API 服务，而每个 Child-Cell 运行除 Nova-API 以外的全部 Nova 服务，且每个 Child-Cell 运行自己的消息队列、数据库及 Nova-cells 服务，即 Cell 之间的消息队列和数据库都是独立的。Nova Cell 通过将共享的消息队列和数据库拆分为以子 Cell 为单位的独立消息队列和数据库，以分而治之的思想解决大规模计算节点部署时经常出现的消息堵塞和数据库性能问题。在多

Region 大规模 OpenStack 公有云部署中, Cell 是 OpenStack 实现水平大规模扩展的理想方案, 诸如 Rackspace、GoDaddy 和欧洲核子中心 (CERN) 这样的 OpenStack 超级用户都在使用 Cell 来部署 OpenStack。

1. Nova Cell V1

从 Nova Cell 的发展历程来看, Nova Cell 可以分为 Nova Cell V1 和 Nova Cell V2 两个发展阶段。而从部署用户数量和知名度来看, Nova Cell V1 除了 CERN 外几乎没有太多用户使用该功能, 并且社区参与讨论和开发的人员也非常有限, 可以说 Cell V1 是个相对冷门的 Nova 功能模块。图 8-20 为 Nova Cell V1 的拓扑架构图, 在 Nova Cell V1 架构中, 不同的计算资源被隔离为多个独立的 Cell, 而在每个 Cell 中均有独立的调度、消息队列和数据库服务, 不同的 Cell 之间可以构成树状结构, 并逐步向下扩展, 因而可以轻松实现 OpenStack 计算节点的规模扩展。每个 Cell 中都有独立的数据库服务、独立的消息队列, 而不同的 Cell 之间则通过 Nova-Cells 服务进行消息传递。在启用 Nova Cell V1 的 OpenStack 集群中, 用户创建虚拟机的请求首先到达顶层的 API Cell, 然后通过 API Cell 的调度算法决定虚拟机应该由哪个 Cell 负责创建, 当某个 Compute Cell 接收到来自 API Cell 的请求后, Compute Cell 将把这个请求通过 Nova-Cells 服务传递给 Cell 中的 Nova-scheduler 服务进行与 Cell 无关的主机调度, Compute Cell 中的 Nova-scheduler 服务收到请求后, 再根据主机资源统计信息决定将虚拟机创建在哪一台物理服务器上^①。

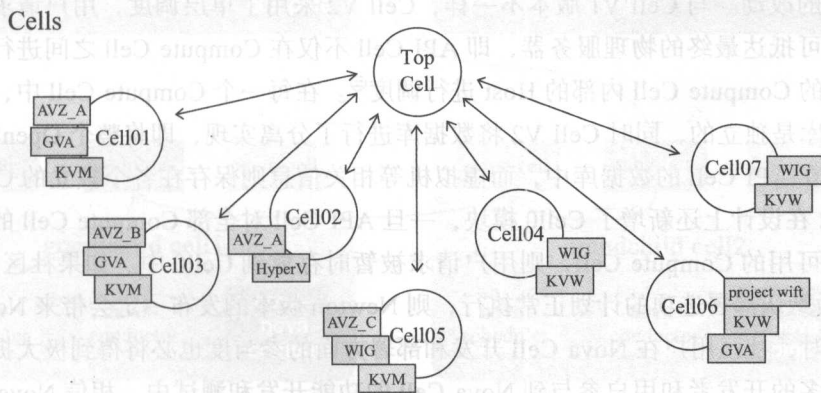


图 8-20 Nova Cell V1 架构图

Nova Cell V1 版本在用户实际部署和社区开发过程中都存在诸多问题和限制。Nova 的网络模块 Nova-network 在设计之初并没有主机集合 (Host Aggregate) 的概念, 其对安全组 (Security Group) 的支持具有一定的局限性, 同时对可用域 AZ 的支持和 Cell 内部的调度功能都有很大的局限性。在 Nova Cell V1 中, 用户请求需要经历两层调度, 即顶层 API Cell 在多个 Compute Cell 之间的调度和各个 Compute Cell 内部的调度, 而在 Nova Cell V1 的内

^① <https://www.ustack.com/news/what-is-nova-cells-v2/>

部 Cell 中，调度所支持的策略和过滤条件都具有一定的局限性。此外，由于在 Nova 项目中，Nova Cell V1 是一个可选组件，默认并不启用，其成熟度和部署率都很低，使用 Nova Cell V1 的用户极为少数，社区对 Nova Cell V1 的开发和关注讨论都严重滞后和不足。虽然社区在 G 版本就引入 Nova Cell 功能，但是直到 Mitaka 版本发行为止，Nova Cell V1 仍然没有得到普及。因此，社区 Nova 团队吸取了 V1 版本的诸多不足和缺陷，对 Nova Cell 的功能结构进行了重新设计和规划，并正在积极推出具有更多优势功能和合理架构的 Nova Cell V2 版本。基于上述原因，Nova Cell V1 被认为是实验性质的功能，且 Nova Cell 的核心团队已经冻结了 Cell V1 的功能，即不再接受关于 Cell V1 的新功能建议，也不会再去修复 Cell V1 中存在的 Bug，目前团队的工作优先级和重心是 Nova Cell V2 的开发设计和 Cell V1 到 Cell V2 的迁移。

2. Nova Cell V2

在 2016 年上半年的 Austin 峰会中，Nova Cell V2 的核心领导成员 Andrew Laski 做了一场关于 Nova Cell V2 及其未来发展方向的介绍^①。Andrew 的演讲透露，在即将发行的 Newton 版本中，Nova Cell 将是默认功能，即用户在部署 OpenStack 时必须部署 Nova Cell V2，这个改变必然会对未来 OpenStack 的部署使用产生影响，并且对于 OpenStack 集群的架构设计、云服务的高可用设计等方面也都将产生深远的影响。Nova Cell V2 的架构如图 8-21 所示，Nova Cell V2 版本充分考虑了 Cell V1 版本的不足，尤其是在 Cell 调度方面从设计上做了根本性的改动。与 Cell V1 版本不一样，Cell V2 采用了单层调度，用户请求只需通过一层调度即可抵达最终的物理服务器，即 API Cell 不仅在 Compute Cell 之间进行调度，还同时对选定的 Compute Cell 内部的 Host 进行调度^②。在每一个 Compute Cell 中，消息队列系统和数据库是独立的，同时 Cell V2 将数据库进行了分离实现，即将整个 OpenStack 的全局信息保存在 API Cell 的数据库中，而虚拟机等相关信息则保存在各个独立的 Cell 中。此外，Cell V2 在设计上还新增了 Cell0 模块，一旦 API Cell 对全部 Compute Cell 的调度都失败或者没有可用的 Compute Cell，则用户请求被暂时存放到 Cell0 中。如果社区关于 Nova Cell V2 成为默认部署选项的计划正常执行，则 Newton 版本的发布一定会带来 Nova 功能上的革新，届时，社区用户在 Nova Cell 开发和部署方面的参与度也必将得到极大提升，而伴随着越来越多的开发者和用户参与到 Nova Cell 的功能开发和测试中，相信 Nova Cell 功能在不久的将来必将走向稳定和成熟，而基于 OpenStack 的大规模可扩展、多区域、容灾高可用部署也会变得越来越简单可行。

对 Nova Cell 功能使用最为成熟和具有丰富经验的当属 OpenStack 超级用户 CERN，尽管 Nova Cell V1 版本对社区而言仅是实验阶段，并且使用 Nova Cell V1 也会失去 OpenStack 的某些功能，但是 CERN 仍然采用 Nova Cell V1 进行了大规模的 OpenStack 集群部

① <https://www.openstack.org/videos/video/nova-cells-v2-whats-going-on>

② <http://docs.openstack.org/developer/nova/cells.html>

署，并成功实现了计算资源的 Region 隔离和基于 Nova Cell 的计算资源统一调度。图 8-22 为 CERN 在瑞士日内瓦（Geneva, Switzerland）和匈牙利布达佩斯（Budapest, Hungary）两地三中心以 Nova Cell V1 模式部署的 OpenStack 集群拓扑。

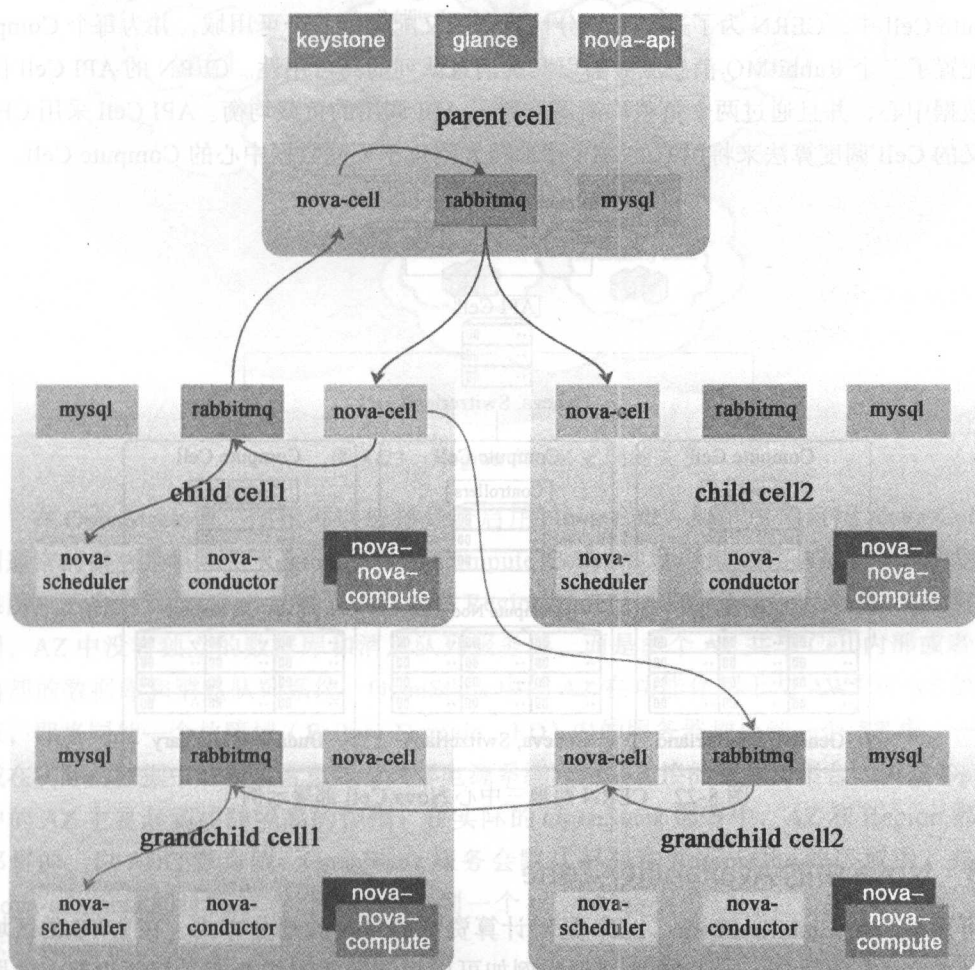


图 8-21 Nova Cell V2 架构

为了应对大型强子对撞机（Large Hadron Collider, LHC）相关实验人员对高计算密度资源的需求，CERN 的技术部门最终部署了基于 RDO 和 Scientific Linux 系统的 OpenStack 云计算集群。通过部署 OpenStack，CERN 简化了对计算资源的管理，并在未增加技术人员的提前下，通过增加 OpenStack 数据中心实现了计算能力的翻倍提升。CERN 两地三中心的 OpenStack 部署模式便采用了 Nova Cell 功能：Nova Cell 模式将 CERN 的 OpenStack 计算资源进行了隔离，从而使得各个数据中心的 OpenStack 集群可以实现透明独立的水平扩展。在 CERN 看来，虽然 Nova Cell 的部署使得 CERN 不能使用 OpenStack 的某些功能，

如安全组和实时迁移功能，并且 CERN 的技术人员也需要在不同数据中心之间手工拷贝类似 Flavors 这样的某些细节信息，但是，Nova Cell 的部署模式确实解决了 CERN 的不同数据中心为用户提供统一公共服务 Endpoint API 的问题和计算节点扩展时的的问题。在图 8-22 中的架构中，CERN 创建了一个 API Cell（父 Cell）和三个 Compute Cell（子 Cell）。在每个 Compute Cell 中，CERN 为了进一步划分计算资源又配置了三个可用域，并为每个 Compute Cell 配置了三个 RabbitMQ 消息服务器以实现消息队列的高可用性。CERN 的 API Cell 位于瑞士数据中心，并且通过两个负载均衡器配置了 API 调用的负载均衡。API Cell 采用 CERN 自定义的 Cell 调度算法来将用户的 API 请求转发到位于不同数据中心的 Compute Cell。

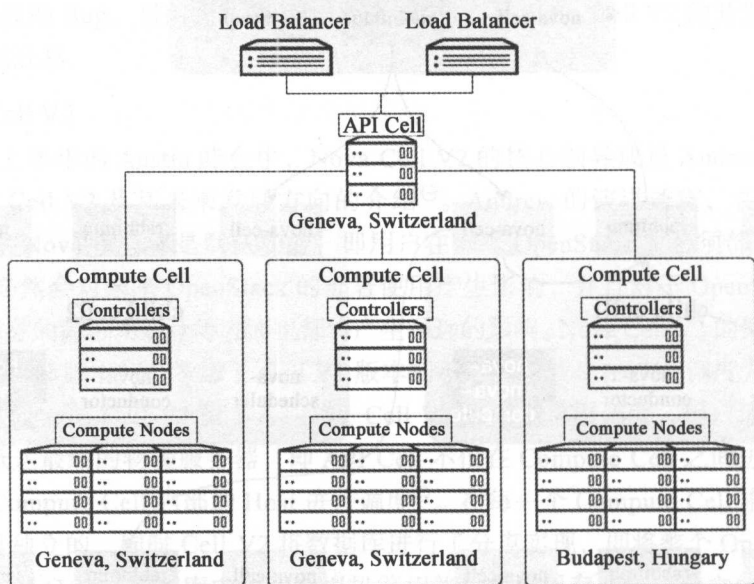


图 8-22 CERN 两地三中心 Nova Cell 部署示例

8.3.3 Nova 中的 Availability Zone

可用域（Availability Zone, AZ）是对计算资源的另一种划分方式，在 AWS 的区域划分设计中，AZ 是对 Region 的再次划分，例如可以把同处相同机架或者相邻机架上的服务器划分到同一个 AZ 中，或者可以把同一个 Region 中彼此独立供电和制冷的机房划分为独立的 AZ。按照 AWS 的解释，划分 AZ 的主要目的是为了提高容灾性和提供廉价的故障隔离服务。用户在不同的 Region 之间做出选择时，主要考虑的是距离自己最近的 Region 或者 Region 所处国家/地区的政策法规是否满足自己服务器和业务系统运行的需求。通常，在没有特殊考虑的情况下，距离用户最近的 Region 应该是首选，例如 AWS 的美国用户，自然会选择离美国近的较近或国内的 Region，而用户在选定 Region 后，还可以选择将自己的实例部署在不同的 AZ 中，选择不同 AZ 部署实例的主要原因在于防止某个 AZ 故障时位于其上的所有实例同时宕机。图 8-23 为 AWS 中 Region 与 AZ 之间的关系示例图。

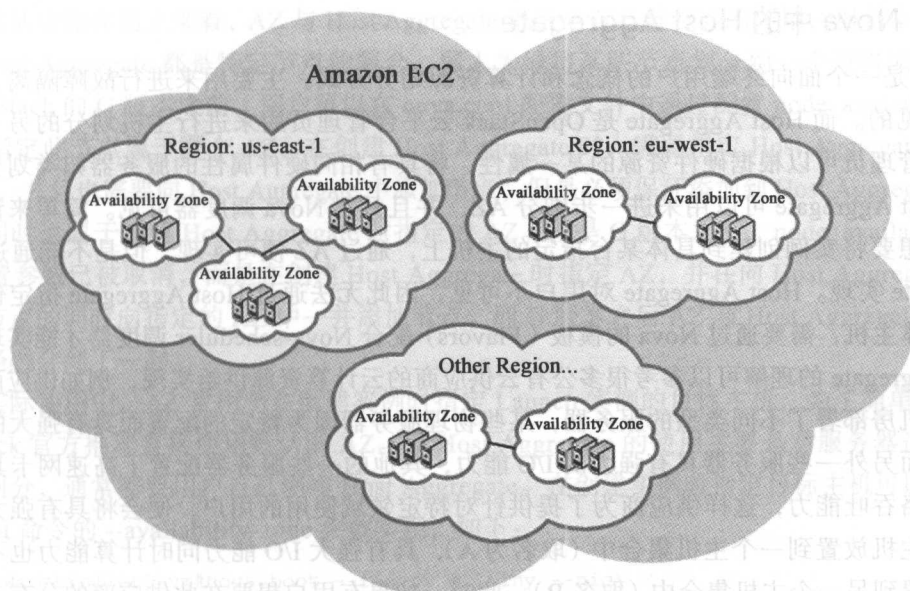


图 8-23 Region 与 AZ 之间的关系

在 OpenStack 中，用户可以选择是否启用 Nova Cell。如果用户启用 Nova Cell 功能，则通常的做法是将单个 Region 部署为 Compute Cell，然后再到 Cell 中进行 AZ 的划分；如果用户不启用 Nova Cell 功能，则直接在 Region 中进行 AZ 划分即可。此外，AZ 与 Cell 不同，AZ 中没有独立的数据库和消息队列服务器，而是多个 AZ 共享 Cell 内部或者 Region 内部的数据库和消息队列系统。OpenStack 中的 AZ 在功能作用上与 AWS 对 AZ 的定义类似，即将同处一个故障域（Failure Domain，FD）内的服务器划分到一个 AZ 中，一个故障域在实际的数据中心中通常意味着共享电源系统和网络连接的服务器集合，因此 OpenStack 中的 AZ 主要起到故障隔离的作用。在实际的 OpenStack 部署中，AZ 和 Region 都是默认部署的，以 RDO 源为例，OpenStack 服务会默认部署在 RegionOne 的区域中，并且部署 Nova-compute 的计算节点会被初始化到一个称为 Nova 的 AZ 中。

从功能实现而言，OpenStack 中的 AZ 功能主要是基于 Nova 的调度服务 Nova-scheduler 来实现的。AZ 对用户是可见的，因此用户在创建实例时，可以显式告知 Nova 应该将实例创建在哪个 AZ 中，当 Nova-scheduler 接收到 AZ 参数后，便会将创建实例的请求调度到指定的 AZ 主机集中，并将实例创建在此 AZ 中的某台主机上，如图 8-24 所示。

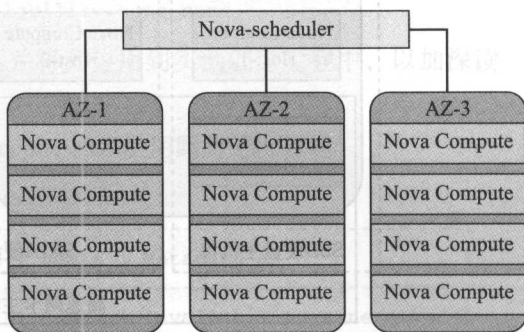


图 8-24 OpenStack 中的可用域 AZ

8.3.4 Nova 中的 Host Aggregate

AZ 是一个面向终端用户的概念和计算资源划分方式，主要用来进行故障隔离，是对用户可见的。而 Host Aggregate 是 OpenStack 云平台管理员用来进行主机划分的另一种方式，即管理员可以根据硬件资源的某一属性，将具有相同硬件属性的服务器归类划分的方式。Host Aggregate 可以用来进一步细分 AZ，并且只对 Nova 调度器可见。简单来说，如果用户想要将实例创建到具体某台指定的主机上，通过 AZ 便可实现，但是不能通过 Host Aggregate 实现。Host Aggregate 对用户不可见，因此无法通过 Host Aggregate 指定创建实例的具体主机，需要通过 Nova 的模板（Flavors）配合 Nova-scheduler 调度器才能实现。对 Host Aggregate 的理解可以参考很多公有云供应商的云计算资源供给实现，例如供应商的数据中心机房部署了不同类型的服务器，某些物理服务器因为特定的配置而具有强大的计算能力，而另外一些服务器具有强大的 I/O 能力，其他的一些服务器配置了高速网卡具有强大的网络吞吐能力，这样供应商为了提供针对特定领域使用的用户，便会将具有强大计算能力的主机放置到一个主机集合中（取名为 A），具有强大 I/O 能力同时计算能力也不错的主机放置到另一个主机集合中（取名 B）。此时，如果有用户想要在此供应商的公有云上部署一个高性能集群，因高性能集群通常有“胖节点”和“瘦节点”之分（“胖节点”既充当计算节点也充当 IO 节点，而“瘦节点”只做计算节点），这样用户便可根据需求在主机集合 A 中创建实例用作“瘦节点”，在主机集合 B 中创建实例用作“胖节点”。而供应商便可根据用户选择进行差异计费，通常计算能力和 I/O 能力都不错的主机集合 B 收费要高于仅提供计算能力的主机集合 A。当然供应商还可以根据很多硬件参数来设置 Host Aggregate，例如使用固态硬盘的机器，内存超过 32G 的机器等，这些服务器的硬件配置都可以成为供应商设置 Host Aggregate 的标准。Host Aggregate 与 AZ 之间的关系如图 8-25 所示，在图中，根据服务器硬盘所属类型，将 AZ 中的服务器又细分为 SSD 硬盘主机集、SATA 硬盘主机集和高速网卡主机集。

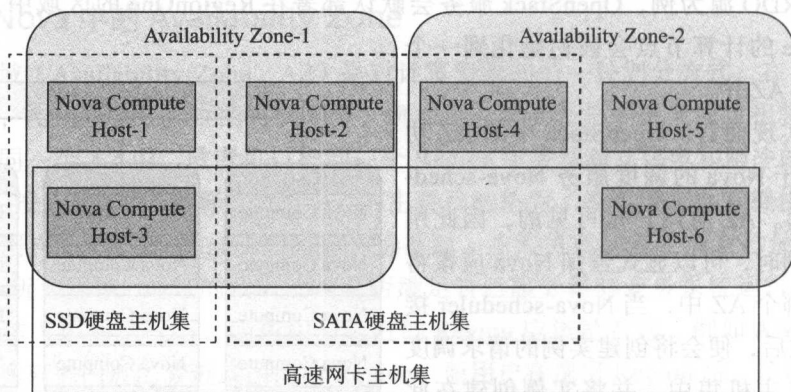


图 8-25 不同硬盘和网卡类型的主机集

虽然从功能作用上来看, AZ 与 Host Aggregate 似乎具有各自不同的作用, 但是本质上, AZ 和 Host Aggregate 都是特定节点的集合, 即人为地将某些节点划分到一个逻辑域里面。在 OpenStack 的 G 版本之前, 用户可以在 nova.conf 配置文件中通过配置 node_availability_zone 来指定此节点属于哪个 AZ, 在创建 Host Aggregate 时, 需要指定 Host Aggregate 属于哪个 AZ, 并且也需要向 Host Aggregate 中添加节点, 但是必须保证添加到 Host Aggregate 中的节点同时也属于创建 Host Aggregate 时指定的 AZ。但是 G 版本之后, node_availability_zone 配置参数已被取消, 而是在创建 Host Aggregate 时指定 AZ, 并在向 Host Aggregate 添加成员节点的同时向指定的 AZ 中一并添加节点, 即 G 版本之后 AZ 与 Host Aggregate 在实现上已经融合到了一起。

在实际使用中, AZ 常用于在创建实例时指定 Launch 实例的目标主机, 为了简单起见, OpenStack 官方推荐直接使用默认的 AZ。而 Host Aggregate 的使用主要是对服务器进行硬件归类划分, 通常需要用户自己创建 Host Aggregate。在创建实例时指定目标主机可以通过 nova boot 命令的 --availability-zone 参数实现, 如下:

```
root@controller1-vm#nova boot --flavor m1.tiny --nic\
net-id=f856c3b6-e959-4698-bcdf-83ba776ac46e --security-group default\
--image cirros-0.3.4-x86_64 --key-name admin-key --availability-zone\
nova:compute-1 admin-instance1
```

上述命令将实例 admin-instance1 创建在名为 nova 的默认 AZ 中, 并且指定了实例应该在可用域 nova 中的 compute-1 主机上创建, 命令执行后, 主机 compute-1 上将会新增一个虚拟机实例。Host Aggregate 的创建命令语法如下:

```
root@controller1-vm:~# nova help aggregate-create
usage: nova aggregate-create <name> [<availability-zone>]
    Create a new aggregate with the specified details.
    Positional arguments:
    <name>                Name of aggregate.
    <availability-zone>   The availability zone of the aggregate (optional).
```

即 Host Aggregate 的创建语法为:

```
nova aggregate-create aggregate_name availability-zone_name
```

下面, 通过几个步骤来创建 Host Aggregate, 并将其应用到实例创建过程中, 以加深读者对 Host Aggregate 的理解。

1) 首先创建一个 Host Aggregate, 并命名为 ceph, AZ 使用默认的 nova。

```
root@controller1-vm#nova aggregate-create ceph nova
```

Id	Name	Availability Zone	Hosts	Metadata
1	ceph	nova		'availability_zone=nova'

2) 向 Host Aggregate 中添加节点。

向 Host Aggregate 添加节点的命令语法为:

```
nova aggregate-add-host aggregate_ID host_name
```

向名为 ceph 的 Host Aggregate 中添加两个主机节点, 分别是 ceph1 和 ceph2:

```
root@controller1-vm#nova aggregate-add-host 1 ceph1
```

```
root@controller1-vm#nova aggregate-add-host 1 ceph2
```

查看添加节点后的 Host Aggregate 状态:

```
root@controller1-vm#nova aggregate-details 1
```

Id	Name	Availability Zone	Hosts	Metadata
1	nova	nova	'ceph1', 'ceph2'	'availability_zone=nova'

可以看到 Host Aggregate 中新增了两个主机节点 ceph1 和 ceph2, 同时有一个默认的元数据信息 “availability_zone=nova”。元数据信息很重要, 在后面的 Host Aggregate 配置中将会使用到元数据信息, 元数据信息是可以自定义的, 用户可以根据实际需要进行设置。

3) 为 Host Aggregate 设置 metadata。

设置元数据信息的命令语法为:

```
nova aggregate-set-metadata aggregate_ID key=value
```

设置元数据信息的 Key-value 键值对为 “ceph=true”:

```
root@controller1-vm#nova aggregate-set-metadata 1 ceph=true
```

```
Metadata has been successfully updated for aggregate 1.
```

Id	Name	Availability Zone	Hosts	Metadata
2	instance_ceph	nova	'ceph1', 'ceph2'	'availability_zone=nova', 'ceph=true'

4) 创建一个 flavor, 为此 flavor 设置与第 3 步相同的 metadata。

Nova 的 flavor 创建语法命令为:

```
nova flavor-create name id ram disk vcpu
```

Nova 的 flavor 元数据设置命令语法为:

```
nova flavor-key flavor_name set key=value
```

创建名为 m1.ceph, ID 为 7 的 flavor, 同时为其设置元数据键值对 “ceph=true”:

```
root@controller1-vm#nova flavor-create m1.ceph 7 512 5 1
```

```
root@controller1-vm#nova flavor-key 7 set\
aggregate_instance_extra_specs:ceph=true
```

查看 m1.ceph 的 flavor 详细信息:

```
nova flavor-show m1.ceph
```

Property	Value
OS-FLV-DISABLED:disabled	False
OS-FLV-EXT-DATA:ephemeral	0
disk	5
extra_specs	{"aggregate_instance_extra_specs:ceph": "true"}
id	7
name	m1.ceph
os-flavor-access:is_public	True
ram	512
rxtx_factor	1.0
swap	
vcpus	1

5) 修改配置文件 nova.conf 中的 host-filter 变量值。

//修改前

```
scheduler_default_filters=RetryFilter,AvailabilityZoneFilter,RamFilter,ComputeFilter,ComputeCapabilitiesFilter,ImagePropertiesFilter,ServerGroupAntiAffinityFilter,ServerGroupAffinityFilter
```

//修改后

```
scheduler_default_filters=AggregateInstanceExtraSpecsFilter,RetryFilter,AvailabilityZoneFilter,RamFilter,ComputeFilter,ComputeCapabilitiesFilter,ImagePropertiesFilter,ServerGroupAntiAffinityFilter,ServerGroupAffinityFilter
```

通过以上五个步骤, Host Aggregate 即配置完成。因为修改了 Nova 的配置文件, 所以需要将 Nova 服务重新启动以便配置生效。现在, 如果需要将实例部署到 ceph 这个主机集中, 只要在创建实例时指定使用 m1.ceph 这个模板即可, Nova-scheduler 会自动将实例创建请求调度到 ceph 主机集中, 因为这里 ceph 主机集中有 ceph1 和 ceph2 两个主机, 所以用户实例将会创建在其中一个主机上, 实例创建命令如下:

```
root@controller1-vm#nova boot --flavor m1.ceph --image cirros-0.3.4-x86_64 --nic
net-id=37e629c8-b24f-4a31-a623-9b78f731296d --security-group default --key-
name admin-key admin-instance2
```

8.4 Nova Hypervisor 配置概述

8.4.1 虚拟化与 Hypervisor 概述

Hypervisor 是运行在物理服务器和操作系统之间的软件层, 主要作用在于调度客户机

系统对共享物理硬件资源的使用请求，是全部虚拟化技术的基石，同时也是云计算的核心基础。由于 Hypervisor 允许多个操作系统和应用共享一套基础物理硬件资源，因此也将它看作虚拟环境中的“元”操作系统，而虚拟客户机系统则可以看作 Hypervisor 这个“元”操作系统的进程。通常，各种类型的 Hypervisor 都可以协调访问服务器上的所有物理设备和虚拟机，因而也将 Hypervisor 称为虚拟机监视器（Virtual Machine Monitor, VMM）。Hypervisor 是所有虚拟化技术的核心。对虚拟客户机进行资源隔离或共享调度，并支持非中断的多工作负载迁移是各种 Hypervisor 所具备的共有功能。当通过 Hypervisor 为虚拟客户机分配指定资源后，一旦服务器启动并执行 Hypervisor 时，其便会给已创建的每一台虚拟机分配适量的内存、CPU、网络和磁盘资源，并加载所有虚拟客户机的操作系统，开始对硬件资源进行调度响应以供虚拟客户机使用，同时对客户机系统进行隔离。在 Hypervisor 对硬件资源的调度下，用户对虚拟客户机的体验和使用与运行在普通物理机上的系统并无差异。图 8-26 为 Hypervisor 虚拟化前后的物理服务器与操作系统运行情况。在传统应用下，物理服务器只能部署单操作系统，应用程序部署在操作系统中，通过操作系统来进行硬件资源的调用；而在虚拟化环境下，物理服务器的 CPU、内存、硬盘和网卡等硬件资源被虚拟化并受 Hypervisor 的调度，多个操作系统在 Hypervisor 的协调下可以共享这些虚拟化后的硬件资源，同时每个操作系统又可以保存彼此的独立性。

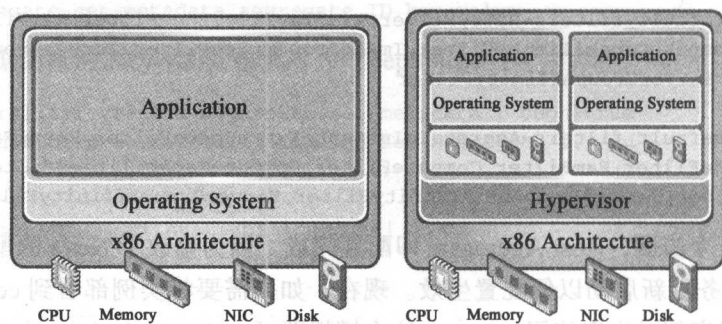


图 8-26 Hypervisor 虚拟化前后对比

在 Hypervisor 虚拟化环境下，部署在物理服务器上的系统称为 Host OS，而部署在 Hypervisor 上的虚拟机操作系统称为 Guest OS。根据 Hypervisor 所处层次的不同和 Guest OS 对硬件资源的不同使用方式，Hypervisor 虚拟化被分为两种类型，即 Bare-metal 虚拟化方式（“裸机”虚拟化）和 Host OS 虚拟化方式（基于操作系统的虚拟化）。

Host OS 类型将 Hypervisor 虚拟化层安装在传统的操作系统中，典型的使用方式如 VMware Workstation 和 VirtualBox，此外，QEMU 和 WINE 也属于此类型，虚拟化软件以应用程序进程形式运行在 Windows、Linux 等主机操作系统中，而除了虚拟化软件外，Host OS 中还可以运行其他的应用程序，如图 8-27a 所示。Bare-metal 类型的 Hypervisor 虚拟化环境中无须完整的 Host OS，而是直接将 Hypervisor 部署在裸机上并将裸机服务器的硬件资

源虚拟化，也可以将 Hypervisor 理解为仅对硬件资源进行虚拟和调度的薄操作系统，其并不提供常规 Host OS 的功能，Bare-metal 类型的 Hypervisor 如图 8-27b 所示，常见的裸机类型 Hypervisor 有 IBM 的 PowerVM、VMware 的 ESX Server、Citrix 的 XenServer、Microsoft 的 Hyper-V 以及开源的 KVM 等虚拟化软件。

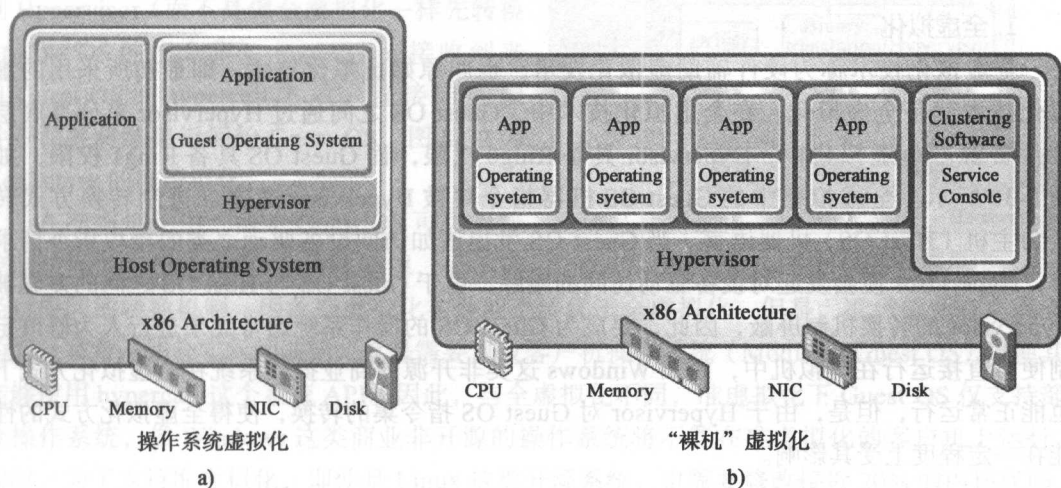


图 8-27 Host OS 与 Bare-metal 类型的 Hypervisor

基于上述两种类型 Hypervisor，虚拟化技术又可以分为全虚拟化 (Full Virtualization, FV)、准虚拟化 (Para Virtualization, PV) 和主机操作系统虚拟化 (Host OS Virtualization)，其中 PV 和 FV 主要是基于 Bare-metal 类型 Hypervisor 的虚拟化技术，而主机操作系统虚拟化主要是基于 Host OS 类型 Hypervisor 的虚拟化技术。此外，除了基于传统 Hypervisor 技术的虚拟化，还有基于最近极为火热的容器技术的虚拟化，虚拟化技术的分类如图 8-28 所示。

虚拟化技术中一个关键性的难题便是物理 CPU 的虚拟化。在 X86 架构的服务器体系中，存在一种称为保护环 (Protection Ring) 或层级保护域 (Hierarchical Protected Domain) 的 CPU 指令特权架构，如图 8-29 所示。运行在 X86 架构上的操作系统 (Operating System, OS) 被设计为具有直接访问和控制硬件资源的权限。X86 架构使用 0、1、2 和 3 四个权限 Ring 来控制管理硬件访问权限，其中用户应用程序运行在 Ring3 权限中。OS 因为需要直接访问物理硬件资源而在 Ring0 中执行。X86 架构下的这种层级授权机制也会被应用到虚拟机中。在虚拟环境中，Guest OS 也需要具有 Ring0 权限才能访问硬件资源，但在实际中 Guest OS 却不能像传统 Host OS 一样直接获取 Ring0 权限，而是需要通过一系列

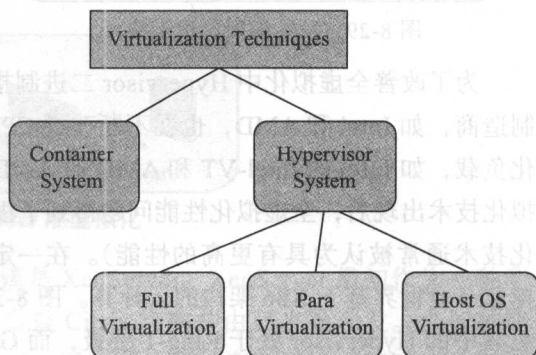


图 8-28 虚拟化技术分类

的复杂方法才能获取 Ring0 权限，这些复杂的方法需要通过 Hypervisor 来完成，正是 CPU 指令赋权方法的复杂性，最终使得 X86 架构服务器底层虚拟化变得极为复杂。

CPU 虚拟化的关键就在于如何对 Guest OS 的请求指令赋予不同层级的权限，而全虚拟化和准虚拟化也正是基于这种对 Guest OS 指令不同的赋权方法来进行划分的。

1. 全虚拟化

全虚拟化技术称为硬件辅助虚拟化技术，也叫原始虚拟化技术，即最初所采用的虚拟化技术就是全虚拟化。在全虚拟化技术中，Guest OS 之间通过 Hypervisor 来分享底层硬件资源。全虚拟化中，Hypervisor 具备 Ring0 权限，而 Guest OS 具备 Ring1 权限，如图 8-30 所示。全虚拟化中的 Guest OS 机器指令集被 Hypervisor 通过二进制转换方式转换为主机（Host OS）机器语言。当 Guest OS 发出诸如访问设备驱动之类的授权指令请求时，Hypervisor 便会发起对设备驱动访问的跟踪，由于 Guest OS 与 Host OS 之间的差异被 Hypervisor 的转换机制屏蔽，因此，要成为 Guest OS 的操作系统通常无须进行人为修改定制便可直接运行在虚拟机中，而如 Windows 这类非开源的商业操作系统在全虚拟化方式下也能正常运行，但是，由于 Hypervisor 对 Guest OS 指令集转换，使得全虚拟化方式的性能在一定程度上受其影响。

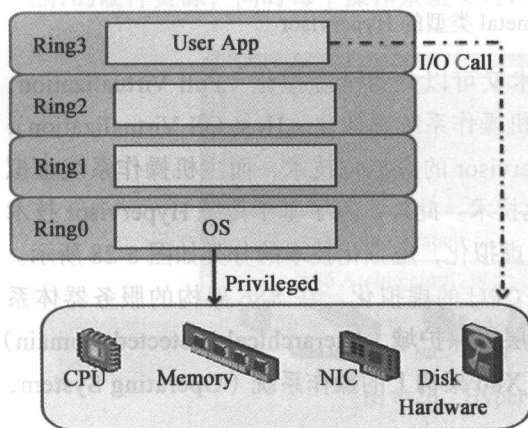


图 8-29 X86 权限层级架构

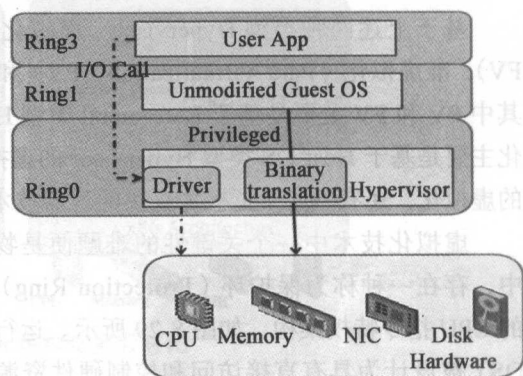


图 8-30 全虚拟化

为了改善全虚拟化中 Hypervisor 二进制指令集转换对 Guest OS 性能的影响，CPU 芯片制造商，如 Intel 和 AMD，也在不断改进 CPU 对虚拟化的支持，从而降低硬件层面的虚拟化负载，如 Intel 的 Intel-VT 和 AMD 的 AMD-V 虚拟化技术，尤其是在 Intel 的 Intel-VT 虚拟化技术出现后，全虚拟化性能问题得到了极大改善，并能与准虚拟化技术相媲美（准虚拟化技术通常被认为具有更高的性能）。在一定程度上，可以说正是 Intel 的 Intel-VT 技术支撑起了全世界基于 X86 架构的云计算。图 8-31 为基于 Intel-VT 虚拟化技术的全虚拟化，注意其中的 Hypervisor 处于 Ring-1 层级，而 Guest OS 处于 Ring0 层级，这与图 8-30 是不一样的。

2. 准虚拟化

准虚拟化也称为半虚拟化，在准虚拟化中，当 Guest OS 执行授权指令时，指令会被一种称为 hypercall 的系统 API 调用传递到 Hypervisor（而不是像全虚拟化一样先转换为 Host OS 的指令集），Hypervisor 接收到来自 Guest OS 的 hypercall 之后，直接访问硬件资源并将结果返回给 Guest OS，图 8-32 为 XenServer 的准虚拟化。

在准虚拟化中，由于 Guest OS 可以直接控制访问如 CPU 和内存等硬件资源，而不需经过中间的转换机制，因此准虚拟化在性能上要优于全虚拟化。但是，准虚拟化有个固有的而且非常不切合实际的缺陷，就是需要更改客户机操作系统（Modified Guest OS）以使其能够使用 hypercall 这个系统 API。因此，与全虚拟化不同，准虚拟化下 Guest OS 仅支持部分操作系统，像 Windows 这类商业非开源的操作系统将不能在准虚拟化的客户机上运行。同时，为了支持准虚拟化，即使是 Linux 这类开源系统，也需要修改接近 20% 的内核代码。总体来说，准虚拟化不仅支持的 Guest OS 有限，而且对 Guest OS 进行内核级别的修改也需要专业的技术能力和大量的修改工作。

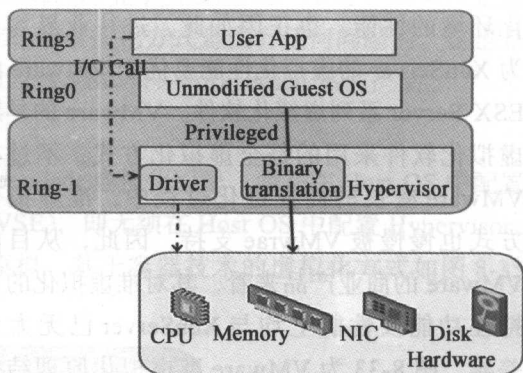


图 8-31 基于 Intel-VT 的全虚拟化

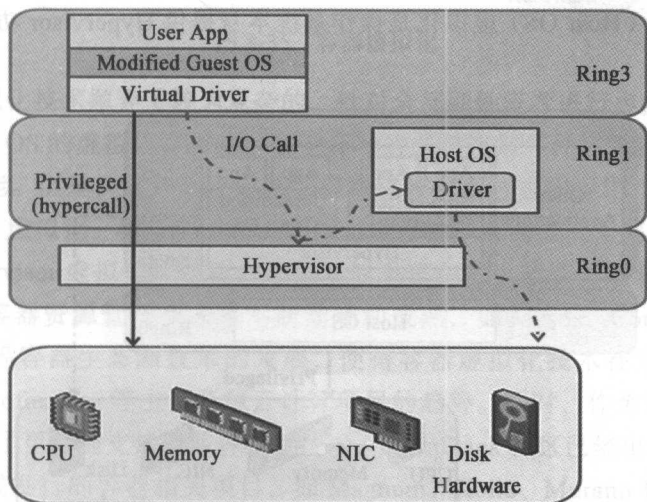


图 8-32 XenServer 准虚拟化

在准虚拟化领域做得最深和最为成熟的应该是 XenServer。XenServer 最初作为一种开源的虚拟化方案而被很多厂商和用户使用，之后被 Citrix 收购并由其主导开发。由于早期 CPU 对虚拟化功能的支持很有限，全虚拟化方式占用过多的硬件资源导致虚拟客户机性能

很差，因此，XenServer 在最初的虚拟化方案中便一直主导使用准虚拟化方式以提升虚拟化环境的性能。也正因如此，过往业界均认为 XenServer 的虚拟化性能要优于 VMware 的 ESX Server 系列虚拟化软件。VMware 的早期虚拟化软件采用的是全虚拟化方式，不过在 VMware 漫长的商业进化过程中，准虚拟化方式也慢慢被 VMwrae 支持，因此，从目前 VMware 的商业产品来看，其对准虚拟化的支持在功能及性能上均与 XenServer 已无太多差异，图 8-33 为 VMware 准虚拟化原理结构图。同时，由于 VMwrae 多年来对全虚拟化的支持和发展，加之 Intel 和 AMD 对 CPU 芯片虚拟化技术的大力改进和支持，在全虚拟化几乎与准虚拟化并无差异的情况下，VMware 的产品以其支持几乎全部客户机操作系统和无须进行任何客户机系统更改的简单易用性，占领了市场近 80% 的虚拟化份额，直到 OpenStack 的出现，用户在虚拟化方案的选取时才真正多出了可选的考虑，而目前的 VMware 仍然是 OpenStack 在私有云领域的强大竞争者。

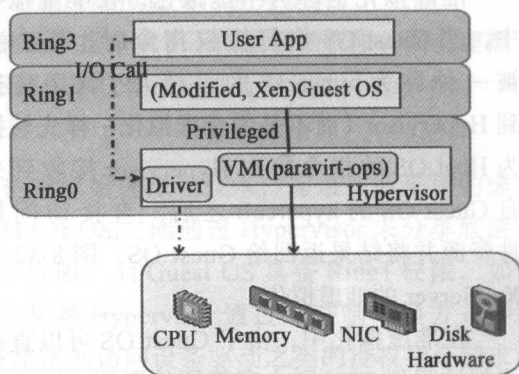


图 8-33 VMware 准虚拟化

3. 操作系统虚拟化

主机操作系统 (Host OS) 虚拟化是操作系统本身提供 Hypervisor 功能的虚拟化方式，如图 8-34 所示。

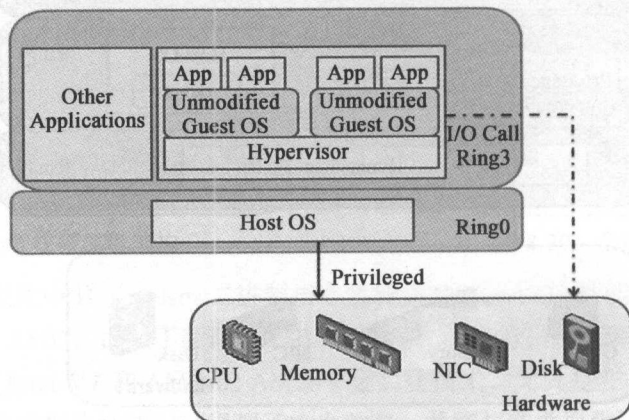


图 8-34 Host OS 虚拟化

虽然主机操作系统虚拟化也可以看作一种有效的虚拟化方式，但是主要应用于虚拟化相关的开发实验环境中。在服务器虚拟化中，由于其虚拟机实例之间不成熟的资源管理，以及性能和安全方面的问题，几乎很少使用主机操作系统虚拟化方式，例如，当运行

Hypervisor 的 Host OS 出现安全问题时，全部 Guest OS 都会受到影响。不过，如果仅是想要在个人 PC 上运行多个操作系统，使用 Host OS 的虚拟化方式是没有任何问题的，这也是 Host OS 虚拟化最常用的方式，其中最为流行的当属 VMware Workstation 系列产品。

4. 容器虚拟化

基于容器的虚拟化技术并不依赖传统的 Hypervisor 虚拟化引擎，而是在 Host OS 中配置虚拟服务器环境（Virtual Server Environment, VSE），即无须在 Host OS 中配置 Hypervisor，并且在容器虚拟化中，仅有 Guest OS 被仿真模拟。基于容器技术的虚拟化方式如图 8-35 所示。

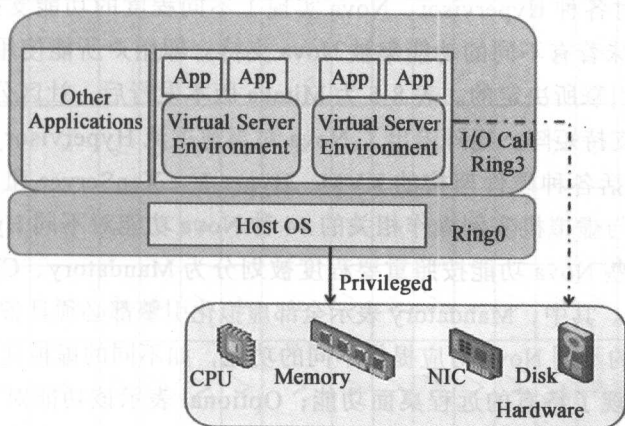


图 8-35 容器虚拟化

由于容器虚拟化技术抛弃了较为复杂的，针对全部硬件资源进行虚拟化的 Hypervisor，且是仅针对 Guest OS 的虚拟化，因此，基于容器的虚拟化是一种“更轻”的虚拟化方式，也具有更好的性能。但是，这种基于容器的虚拟化技术在过去很少用于服务器虚拟化中，因为就如 Host OS 虚拟化一样，在过去，容器虚拟化技术对虚拟机实例之间的资源和管理都不如 Hypervisor 虚拟化。然而，这一切似乎正在发生改变。随着 Docker 等容器技术的发展，新的容器资源管理技术也不断涌现和成熟，如 Mesos、CoreOS、Kubernetes、Swarm 和 Rocket 等容器生态圈技术的发展，使得容器虚拟化技术在发展势头上有赶超 OpenStack 和 Cloudfoundry 等主流开源云计算项目的趋势。同时，作为以大帐篷形式发展的 OpenStack，为了更好地支持和兼容容器技术，OpenStack 社区已经出现很多正在孵化和已经被接受并正式发行的容器相关项目，如 Magnum、Kolla、Murano 和 Nova-lxd 等。随着虚拟化和云计算技术的发展，尤其是各种开源云计算项目对容器的支持，轻量级的容器虚拟化技术协同 OpenStack 在未来引领数据中心已是大势所趋。

8.4.2 Nova Hypervisor 归类支持

OpenStack 的 Nova 项目支持很多主流的 Hypervisor，尽管官方默认和推荐的是

KVM 虚拟化引擎，但是用户可以在 Nova 项目中使用不局限于一个的 Hypervisor，正因如此，基于 VMware 虚拟化的用户可以将其与 OpenStack 进行整合，即可以继续使用 VMware 提供的虚拟化引擎并使用 OpenStack 来部署云计算集群。在多数情况下，为了便于统一管理，推荐在整个 OpenStack 集群中使用单一的 Hypervisor 虚拟化引擎。如果用户根据自己的生产环境准备混合使用多种 Hypervisor，则 OpenStack 提供了 ComputeFilter 和 ImagePropertiesFilter 两种调度方式来在不同的 Hypervisor 之间创建虚拟机实例。

目前为止，Nova 自带很多虚拟化引擎（Virt Driver），每一种虚拟化引擎都对应着不同的 Hypervisor。针对各种 Hypervisor，Nova 实现了不同程度的功能支持，因此选择不同的虚拟化引擎就意味着有不同的功能集被 Nova 支持，即用户所能使用的 Nova 功能是由用户选取的虚拟化引擎所决定的。表 8-5 为 Mitaka 版本发行后，社区公布的 Nova 对不同虚拟化引擎的功能支持矩阵。矩阵给出了 Nova 对当前主流 Hypervisor 虚拟化引擎的功能支持情况，其中包括各种硬件架构的 KVM、Hyper-V、XenServer 和 VMware 等虚拟化引擎，同时列出了与虚拟机实例操作相关的 44 种 Nova 功能对不同 Hypervisor 虚拟化引擎的支持情况。这些 Nova 功能按照重要程度被划分为 Mandatory、Choice、Optional 和 Condition 四个类别。其中，Mandatory 表示全部虚拟化引擎都必须具备的功能；Choice 表示根据虚拟化引擎的不同 Nova 对应提供不同的功能，如不同的虚拟化引擎根据自己特有的虚拟桌面协议实现了特有的远程桌面功能；Optional 表示该功能对于不同的虚拟化引擎来说都是可选的，Nova 对不同的虚拟化引擎可以选择支持或不支持该功能；Condition 则表明此功能重要程度取决于另外的功能，如用户选择了 Nova 的块存储支持功能，则基于 iSCSI 块存储功能就是各个虚拟化引擎必须（Mandatory）支持的功能，否则该功能是可选的（Optional）。从表 8-5 中可以看到，目前社区认定的 Nova 对各个虚拟化引擎都必须支持的功能有四个，分别是客户机实例状态监控查看功能、launch 实例功能、shutdown 实例功能和镜像存储功能。这四个功能在列出的虚拟化引擎中全部得到了 Nova 的支持。此外，从 Nova 支持功能最多的虚拟化引擎来看，当属基于 X86 架构的 KVM 虚拟化引擎，这也是 OpenStack 默认和官方推荐的虚拟化引擎。当然，用户还可以选择 Hyper-V、VMware 和 XenServer 的虚拟化引擎，但是在选用之前，必须对照表 8-5 功能支持矩阵以核实自己需要的某些 Nova 功能是否被选取的虚拟化引擎支持。如要部署高可用 OpenStack 集群，使得虚拟机在宿主机故障时自动迁移（Evacuate 功能），则必须选用 KVM 虚拟化引擎。

对于 OpenStack 的终端用户而言，尤其是准备用于生产系统的用户，仅知道如何选取虚拟化引擎才可获得预需的 Nova 功能并不能消除用户对 Nova 功能稳定性的忧虑。为此，社区对 Nova 的功能进行了归类并对这些功能集在不同虚拟化引擎中的成熟度给出了参考，如表 8-6 所示。

表 8-5 Nova 对 Hypervisors 的功能支持矩阵

Feature	Status	Hyper-V	Ironic	Libvirt KVM (ppc64)	Libvirt KVM (s390x)	Libvirt KVM (x86)	Libvirt LXC	Libvirt QEMU (x86)	Libvirt Virtuozzo CT	Libvirt Virtuozzo VM	Libvirt Xen	VMware vCenter	Xen Server
Attach block volume to instance	optional	✓	×	✓	✓	✓	×	✓	×	✓	✓	✓	✓
Detach block volume from instance	optional	✓	×	✓	✓	✓	×	✓	×	✓	✓	✓	✓
Set the host in a maintenance mode	optional	×	×	×	×	×	×	×	×	×	×	×	✓
Evacuate instances from a host	optional	?	?	?	✓	✓	?	?	×	×	?	?	?
Rebuild instance	optional	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Guest instance status	mandatory	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Guest host status	optional	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Live migrate instance across hosts	optional	✓	×	✓	✓	✓	×	✓	×	×	✓	×	✓
Launch instance	mandatory	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Stop instance CPUs (pause)	optional	✓	×	✓	✓	✓	✓	✓	×	✓	✓	×	✓
Reboot instance	optional	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Rescue instance	optional	✓	×	✓	✓	✓	×	✓	✓	✓	✓	✓	✓
Resize instance	optional	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓
Restore instance	optional	✓	×	✓	✓	✓	×	✓	✓	✓	✓	✓	✓
Service control	optional	×	×	✓	✓	✓	×	✓	×	×	×	✓	✓
Set instance admin password	optional	×	×	×	×	✓	×	✓	×	×	×	×	✓
Save snapshot of instance disk	optional	✓	×	✓	✓	✓	×	✓	✓	✓	✓	✓	✓
Suspend instance	optional	✓	×	✓	✓	✓	×	✓	✓	✓	✓	✓	✓
Swap block volumes	optional	×	×	✓	✓	✓	✓	✓	×	✓	✓	×	×
Shutdown instance	mandatory	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Trigger crash dump	optional	×	×	✓	✓	✓	×	✓	×	×	×	×	×

(续)

Feature	Status	Hyper-V	Ironic	Libvirt KVM (ppc64)	Libvirt KVM (s390x)	Libvirt KVM (x86)	Libvirt LXC	Libvirt QEMU (x86)	Libvirt Virtuoizzo CT	Libvirt Virtuoizzo VM	Libvirt Xen	VMware vCenter	Xen Server
Resume instance CPUs (unpause)	optional	√	×	√	√	√	√	√	√	√	√	×	√
Auto configure disk	optional	√	×	×	×	×	×	×	×	×	×	×	√
Instance disk I/O limits	optional	×	×	√	√	√	×	√	×	×	×	×	×
Config drive support	choice	√	√	×	√	√	√	√	×	√	√	√	√
Inject files into disk image	optional	×	×	×	×	√	×	√	×	×	×	×	√
Inject guest networking config	optional	×	×	×	×	√	×	√	×	×	×	√	√
Remote desktop over RDP	choice	√	×	×	×	×	×	×	×	×	×	×	×
View serial console logs	choice	√	×	×	√	√	×	√	×	×	√	√	√
Remote interactive serial console	choice	√	×	?	√	√	?	?	×	×	?	×	×
Remote desktop over SPICE	choice	×	×	×	×	√	×	√	×	×	×	×	×
Remote desktop over VNC	choice	×	×	×	×	√	×	√	√	√	√	√	√
Block storage support	optional	√	×	√	√	√	√	√	×	√	√	√	√
Block storage over fibre channel	optional	×	×	×	√	√	√	√	×	√	√	×	×
Block storage over iSCSI	condition	√	×	√	√	√	√	√	×	√	√	√	√
CHAP authentication for iSCSI	optional	√	×	√	√	√	√	√	×	√	√	×	√
Image storage support	mandatory	√	√	√	√	√	√	√	√	√	√	√	√
Network firewall rules	optional	×	×	√	√	√	√	√	√	√	√	×	√
Network routing	optional	×	√	×	√	√	√	√	√	√	√	√	√
Network security groups	optional	×	×	√	√	√	√	√	√	√	√	√	√
Flat networking	choice	√	√	√	√	√	√	√	√	√	√	√	√
VLAN networking	choice	×	×	√	√	√	√	√	√	√	√	√	√
uefi boot	optional	×	√	×	×	√	×	√	×	×	×	×	×

表 8-6 Nova 功能分类

Feature	Maturity	Hyper-V CI	IroniC CI	libvirt+kvm (gate)	libvirt+xen	VMware CI	XenServer CI
Create Server and Delete Server	completed	√	?	√	√	√	√
Snapshot Server	completed	?	?	√	√	?	√
Server power ops	completed	√	?	√	√	√	√
Rebuild Server	completed	√	?	√	√	√	√
Resize Server	completed	√	?	√	√	√	√
Volume Operations	completed	√	×	√	√	√	√
Custom disk configurations on boot	completed	√	×	√	√	√	√
Custom neutron configurations on boot	completed	√	×	√	√	√	√
Pause a Server	completed	√	×	√	√	√	√
Suspend a Server	completed	√	×	√	√	√	√
Server console output	completed	√	×	√	√	√	√
Server Rescue	completed	√	×	√	√	√	√
Server Config Drive	completed	√	√	√	√	√	√
Server Change Password	experimental	√	×	√	×	×	√
Server Shelve and Unshelve	completed	√	×	√	√	×	√
NUMA Placement	experimental	?	?	√	×	?	?
GPU Passthrough	experimental	×	?	√	×	×	√

表 8-6 中，成熟度为“completed”的 Nova 功能至少在一种或多种虚拟化引擎中完成了 CI 或 Gate 测试，而成熟度为“experimental”的 Nova 功能仅在一种或多种虚拟化引擎中完成了部分测试。如“Server Config Drive”功能在表 8-9 列出的全部虚拟化引擎中均完成了测试，而针对 NFV 的“NUMA Placement”功能和针对 HPC 的“GPU Passthrough”功能仅在个别虚拟化引擎中完成了部分测试。根据社区的规划，Nova 功能集有 5 个成熟度，分别是未完成（Incompleted）、试验阶段（Experimental）、完成（Completed）、完成且必须（Completed and Required）和过时（Deprecated）。

❑ Incompleted：未完成表明此 Nova 功能集不具备充分的功能来满足用户实际的部署要求。

❑ Experimental：试验阶段的 Nova 功能集可以被用户部署使用，但是在使用过程中一定要非常谨慎，或者对此功能一定要非常熟悉，因为此类功能集很可能还未被 OpenStack 上游开发人员测试过或仅测试了部分功能，因此极可能出现 Bug。

- ❑ **Completed**：某个 Nova 功能集被标记为完成，则说明其具备了完善的 API 文档（概念和 REST 调用的定义），完善的管理员文档，经过了功能正确性的 Tempest 测试，具备实际部署场景所需的完善功能和可靠性，并且该功能不会在短期内被丢弃。
- ❑ **Completed and Required**：当某个功能集成度为完成，并且所有虚拟化引擎都已经支撑此功能，则该功能集将由完成转变为完成且必须。一旦某个功能集进入这个状态，则新的虚拟化引擎要进入 Nova 的上游，就必须具备充分的理由以证明其支持此功能集。被标记为必须的功能是所有虚拟化引擎都要支持的功能，此类成熟度的功能集越多，则表明采用不同虚拟化引擎均可使用的 Nova 功能集越多。
- ❑ **Deprecated**：当社区计划在下一个 Nova 版本中将某个功能集移除时，此功能集就被标记为过时。如果某个 Nova 功能集被标记为完成，则不会被标记为过时，而如果某个功能集在经历了几个发行版本后，仍然还是未完成和实验阶段，则很有可能后续版本中被移除。

在使用 Nova 部署云计算时，用户可以根据实际情况选择不同的虚拟化引擎，但是不同虚拟化引擎对 Nova 的功能支持程度各不相同，而 Nova 的功能集针对不同虚拟化引擎的成熟度也不一样。因此，用户在选用虚拟化引擎时，一定要充分了解自己需要使用到哪些 Nova 功能集，此功能集是否被自己选用的虚拟化引擎支持，如果被支持，此功能集针对选用的虚拟化引擎具有什么样的成熟度。通常，如果是全新搭建基于 X86 架构的 OpenStack 集群，则 KVM 虚拟化引擎是 Nova 社区支持度和成熟度最好的，选用 KVM 可以使用几乎全部的 Nova 功能。特殊情况下，某些用户可能想要将已有的 VMware 或 XenServer 集群集成到 OpenStack 集群中，此时便会有 VMware 的虚拟化引擎需要考虑。此外，对于某些 Nova 功能，如一直处于 Experimental 状态的 Nova Cell V1 功能，尽管有 CERN 这样的超级用户在使用，但是 CERN 有 Tim Bell 等技术领军人物存在，因此在选用处于实验阶段的 Nova 功能集时一定要十分谨慎。

8.4.3 Nova Hypervisor 选取配置

Nova 支持多后端 Hypervisor 驱动，具体选用哪种 Hypervisor，需要用户根据自己的服务器物理架构和 8.4.2 节中的 Nova 功能分类及 Hypervisor 支持矩阵来谨慎选择。对于大多数用户而言，硬件物理架构多为基于 Intel X86 的服务器，可能也有部分企业级用户希望部署基于 IBM Power 架构的 OpenStack 云环境，而 Nova 均支持这两种物理架构。虽然 OpenStack 社区默认和推荐的 Hypervisor 是 KVM，但是 Nova 也支持很多主流的 Hypervisor，如 VMware Vsphere、Xen/XenServer、Hyper-V、LXC、QEMU 和 UML 等。

(1) KVM

KVM (Kernel Virtual Machine) 是基于 Linux 内核的虚拟机，最初由以色列 Qumranet 公司开发，并于 2006 年 10 月问世，但是 2008 年 9 月，Redhat 收购 Qumranet 之后成为了 KVM 新的开发维护者。在 Linux 开源社区，自 Linux 2.6.20 内核之后 KVM 就被集成在各

个 Linux 主要发行版本中,就 Redhat 的 RHEL 系列 Linux 系统而言,从 RHEL6 开始 KVM 便被集成到了内核中,从而取代了 RHEL5 内核中 Xen 的虚拟化地位。基于 KVM 的虚拟化需要硬件的虚拟化扩展支持,如 Intel 的 Inter VT-X 和 AMD 的 AMD-V 技术。此外,KVM 是一种全虚拟化(Full Virtualization)技术,运行在 KVM 上的 Guest OS 无须进行任何修改。从 Linux 内核层面而言,KVM 是 Linux Kernel 中的一个模块,用户可以使用 modprobe 命令加载 KVM 模块,只有加载了 KVM 模块之后,才可以在 Linux 系统中进行虚拟机相关的操作。由于 KVM 对硬件架构的依赖,正常情况下,使用 modprobe 命令加载 KVM 模块后,针对 Intel 和 AMD 的硬件架构,在 Linux 系统中,除了 kvm 模块外,kvm_intel 或 kvm_amd 模块也会被自动加载,对应的模块库文件为 kvm.ko 和 kvm_intel.ko 或 kvm_amd.ko。由于 KVM 模块位于 Linux 内核,同时 KVM 只负责对虚拟机的虚拟 CPU 和内存进行管理调度,而正常运行的虚拟机还需要很多 I/O 接口等外部设备,为了解决这一问题,KVM 社区采用了较为成熟的虚拟化项目 QEMU 来实现 KVM 与虚拟外设的交互,同时使用经过修改的 QEMU 分支 QEMU-KVM 来作为用户空间程序,实现用户与内核 KVM 模块的交互。

在实际使用中,KVM 只负责虚拟机调度和内存管理等工作,而外设相关的任务则由其他 Linux 内核和 QEMU 完成,同时用户可以通过 QEMU-KVM 客户端程序与 KVM 虚拟机进行交互。归结而言,由于 KVM 是一款基于 Linux 内核开发的虚拟化程序,而非一个全新从底层开发的 Hypervisor,因而是一款轻量级的 Hypervisor,并得到了众多 Linux 发行版本的支持。尤其随着云计算的兴起,KVM 以其与 Linux 系统的完美集成,已然成为很多开源虚拟化和云计算解决方案的首选 Hypervisor。

(2) VMware vSphere

VMware vSphere 是 VMware 公司的商业虚拟化产品,是 VMware 公司基于 Linux 内核开发的虚拟化专用系统,通常与 VMware 的 vCenter 集合使用。VMware vSphere 有其对应的单机免费版本,即 VMware vSphere Hypervisor,也称为 ESXi,但是 ESXi 功能有限,不能配合 vCenter 一起使用。商用 VMware vSphere 产品提供了很多丰富的功能,如虚拟机实时迁移的 vMotion 功能、实时虚拟机磁盘迁移的 Storage vMotion 功能、虚拟机高可用(High Availability, HA)功能和物理服务器故障容忍(Fault Tolerance, FT)功能等。在开源云计算 OpenStack 的大潮下,由于 VMware 庞大的虚拟化市场占有率,其通常被看作 OpenStack 的觊觎者。为了更好地支持和加入 OpenStack 生态圈,VMware 也推出了自己的 OpenStack 产品,即 VMware 集成 OpenStack (Vmware Integrated OpenStack, VIO)。VIO 允许用户通过 OpenStack 的 API 调用 vSphere 的功能,而由于 VMware 的加入,OpenStack 社区的 Nova 项目也提供了对 VMware vSphere 的支持。

(3) Xen

Xen 是 2003 年由剑桥大学开发的一款开源 Hypervisor,由于其相对 KVM 问世较早,技术更为成熟,而且具有学院派的严谨风格,因而在早期就拥有大量国内外用户。此外,

硬件虚拟化支持功能通常认为是 2005 至 2006 年左右才分别由 Intel 和 AMD 推出,而在此之前,在全虚拟化技术下,由于 Guest OS 的硬件资源访问指令需要经过复杂的 Hypervisor 进行二进制转换,因此虚拟化性能非常低下,而早期的 Xen 由于其具备准虚拟化功能,可以让 Guest OS 直接访问硬件资源进而极大提高虚拟化性能,因而在很多虚拟化环境中都采用 Xen 虚拟化引擎。然而, Xen 的一大弊端在于不够灵活,使用极为不便,为了让 Guest OS 直接使用硬件资源,不得不对 Guest OS 进行大量内核修改,这通常也意味着像 Windows 这样的系统是无法运行在 Xen 虚拟机上的。作为一个有着成熟技术实力和良好用户基础的 Hypervisor, OpenStack 社区也提供了对 Xen 虚拟化引擎的支持,但是支持力度明显不如 KVM,因此社区也不建议使用 Xen 作为默认的虚拟化引擎。

(4) XenServer

Xen 是社区开源虚拟化项目,而 XenServer 是思杰公司基于 Xen 的商业虚拟化产品,或者可以认为是 Xen 的高级商业版本。XenServer 包含了 Xen 虚拟化引擎以外的产品,通常也被认为是 VMware vSphere 在虚拟化市场上的竞争对手。随着思杰公司的加入, OpenStack 社区也开始支持基于 XenServer 的虚拟化引擎。

(5) Hyper-V

Hyper-V 是微软公司开发的基于 Windows 系统的 Hypervisor。Hyper-V 虚拟化引擎早在 G 版本就被集成到 OpenStack 中,因此,用户只需在运行 Windows 2012 Server 并安装有 Hyper-v 的服务器上部署 OpenStack 的 Nova 代码,即可通过 OpenStack 的 API 部署和管理基于 Hyper-V 的虚拟机。如果用户宿主物理服务器部署的是 Windows 系统,则 Hyper-v 是用户的首选虚拟化引擎。

(6) PowerVM

相对其他虚拟化引擎而言,PowerVM 是一种更为古老和成熟的虚拟化技术,其最初由 IBM 的大机虚拟化技术改进而来,主要是针对 IBM Power 小型机和 AIX 操作系统的虚拟化技术(IBM 早在上世纪 80 年代便开始了大机虚拟化技术的研发和应用)。由于 PowerVM 是一种针对 Power 架构服务器的虚拟化技术,因此很多基于 X86 架构的服务器是不能使用 PowerVM 虚拟化技术的。OpenStack 社区 Nova 项目对 PowerVM 虚拟化引擎的支持,主要还是基于 IBM 庞大的企业级 Power 小型机用户群考虑,而 PowerVM 与 OpenStack 相关的集成工作和代码贡献也主要由 IBM 负责。

(7) QEMU

QEMU 是 Quick EMUlator 的简称,即快速仿真器。QEMU 是一个开源的模拟器项目,在 GNU/Linux 平台被广泛应用。它能够模拟整个系统的硬件,而不管硬件的体系架构是否支持虚拟化。由于其可以在各种系统和硬件架构下模拟虚拟机所需的各种资源或设备,因而 QEMU 通常被用于各种模拟场景中。OpenStack 的 Nova 提供了对 QEMU 的支持,但是除了实验环境或者服务器不支持虚拟化的场景之外,不推荐使用 QEMU 虚拟化引擎,而是采用性能更好的 KVM。

(8) UML

UML 是 User Model Linux 的简称。UML 虚拟机通常用于 Linux 新版本或者内核的功能验证实验,为用户虚拟机提供比实际物理服务器更多的硬件和软件资源。OpenStack 的 Nova 项目提供对 UML 的支持,但是 UML 虚拟机通常仅用于开发实验环境,因此并不推荐在生产环境中部署基于 UML 虚拟化引擎的虚拟机。

(9) LXC

LXC 是 Linux 容器 (Linux Container) 的简称。LXC 并不依赖于具体的某种 Hypervisor,而是操作系统级别的虚拟化。LXC 的主要功能就是在操作系统上为进程提供虚拟的执行环境(一个虚拟机的进程执行环境便成为一个 Container, Container 可以绑定特定的 CPU、内存和 IO 设备等以满足进程运行所需的各种资源)。OpenStack 的 Nova 项目支出 LXC 虚拟机, Nova 对 LXC 的操作控制主要通过 Libvirt 实现。

(10) Bare Metal/Ironic

Bare Metal 即所谓的裸机。裸机表示没有安装任何操作系统和 Hypervisor 的物理服务器(或者虚拟机)。裸机通常采用 PXE 方式进行系统部署,并使用类似于智能平台管理接口 (Intelligent Platform Management Interface, IPMI) 的硬件管理软件进行管理。目前 OpenStack 社区已经将 Nova 的裸机管理功能独立成为 Ironic 项目,并集成到 Kilo 版本中进行发行,但是到 Mitaka 版本发行为止, Ironic 项目的使用率和成熟度依然不是很理想,这与其相对复杂的使用和各厂商硬件难以统一具有一定的关系。

(11) Docker

Docker 是时下最火的新一代容器技术,其基础核心仍然是 LXC。Docker 也是一种基于操作系统的虚拟化技术,主要用于快速部署应用程序,基于 Docker 部署的应用程序能够很好地解决移植时可能存在的程序依赖问题。OpenStack 的 Nova 项目从 Havana 版本便开始引入支持 Docker 的 Hypervisor,除了 Nova 对 Docker 的支持,还有许多与 Docker 相关的项目已经集成发布或正在孵化中, OpenStack 与 Docker 的高度集成也是社区未来的主要工作方向之一。

在虚拟化环境中, Hypervisor 的主要功能就是为虚拟机提供各种所需物理资源的调度。不同的 Hypervisor 位于不同的空间(内核空间或者用户空间),并且其实现和工作原理也不尽相同,因此各种 Hypervisor 都有针对自身的调用接口 API,如 KVM 虚拟化引擎的 qemu-kvm 函数便是用于控制和操作 KVM 虚拟机的 API 函数。在虚拟化领域,为了统一各种 Hypervisor 的 API 接口,开源虚拟机管理接口应用程序 Libvirt 应运而生。Libvirt 是一个软件集合,包括 API 库、后台运行进程 Libvirtd 和命令行工具 virsh,是为了更方便地管理平台虚拟化技术而设计的开源应用程序接口。也可以将 Libvirt 认为是开源虚拟机统一管理接口,如比较常见的虚拟机管理软件 virt-install 和 virt-manager 等在底层使用的都是 Libvirt 应用程序接口。OpenStack 作为一个集大成者的虚拟化管理平台,其 Nova 项目常用的各种 Hypervisor 后端驱动就是 Libvirt,并通过 Libvirt 来管理各种 Hypervisor。目前 Libvirt 支持

的虚拟化方案不仅包括 KVM、Xen、VMware、QEMU 和 VirtualBox 等基于 Hypervisor 的平台虚拟化,还包括 OpenVZ 和 LXC 等基于操作系统的容器虚拟化以及基于用户空间的 UML 虚拟化。Libvirt 作为底层 Hypervisor 与上层应用程序之间的中间驱动层,其常用的功能主要包括虚拟机管理,虚拟设备管理和虚拟机远程控制。

- ❑ 虚拟机管理:虚拟机管理是 Libvirt 最主要的功能,是以基于各种 Hypervisor 所实现的虚拟机为管理对象,提供了对虚拟机的定义、删除、启动、关闭、暂挂、恢复、重建、保存、回滚、快照和迁移等物理机具备和不具备的各种功能。
- ❑ 虚拟设备管理:虚拟机的正常运行需要满足应用程序对设备资源的需求,而不仅仅是实现虚拟机自身的管理。Libvirt 提供了包括 CPU、内存、网卡和磁盘等虚拟设备的管理,并不同程度地支持各种 Hypervisor 下的虚拟设备热插拔功能。对于任何运行 Libvirtd 进程的主机,都可以利用 Libvirt 来管理不同类型的存储,如创建 qcow2、raw、vmdk 等不同格式的镜像,对磁盘设备进行分区,挂载 iSCSI 共享存储等。
- ❑ 虚拟机远程控制:Libvirt 不仅可以管理本机上的 Hypervisor,还可以通过远程连接,管理远程主机上的 Hypervisor。只要主机上运行有 Libvirtd 守护进程,Libvirt 的远程管理程序就可连接到该节点,在认证授权通过后,即可对该主机上的 Hypervisor 进行管理,而本地主机上的全部 Libvirt 功能都可以被远程访问和调用,如在 OpenStack 中进行主机之间的实例迁移时,便会用到 Libvirt 的远程管理功能访问目标节点的 Libvirtd 进程。

因为 KVM 是 OpenStack Nova 项目的默认 Hypervisor,也是使用最为广泛的 Hypervisor,这里主要讲解基于 KVM 的 Hypervisor 在 Nova 中的配置使用,其他 Hypervisor 的详细配置使用方式可以参考 OpenStack 的官方配置参考文档^①。在配置使用 KVM 虚拟化引擎之前,首先要确保物理服务器支持虚拟化功能,并确保主板中的 Intel-VT 或 AMD-V 功能已经打开。对于 RHEL6&7/CentOS6&7 系列的 Linux 系统,通常最小安装之后即可使用 KVM 和 Libvirt 功能(本节讲解的主机系统为 Centos7);对于其他发行版本的 Linux 系统,可以查看 OpenStack 的官方网站^②了解如何安装启用 KVM。如果服务器支持虚拟化并开启了 Intel-VT 或 AMD-V 功能,则系统启动时便会加载 KVM 模块,如果不确定当前主机是否支持虚拟化,可以通过如下命令判断:

```
[root@mitaka ~]# grep -E 'svm|vmx' /proc/cpuinfo
```

如果上述命令有输出,则说明当前主机 CPU 支持虚拟化(仍然需要确保主板 BIOS 中已经开启虚拟化功能);如果上述命令没有任何输出,则需要核实服务器是否支持虚拟化或者主板是否开启虚拟化支持功能。对于没有任何输出的主机,加载 KVM 模块命令运行后

① <http://docs.openstack.org/mitaka/config-reference>

② <http://docs.openstack.org/mitaka/config-reference/compute/hypervisor-kvm.html>

系统会有“不支持 KVM”的提示，此时便不能使用 KVM 虚拟化引擎，而必须改用 QEMU 或 Xen 等其他虚拟化引擎。在支持虚拟化功能的主机中，如果系统没有自动加载 KVM 模块，则可以通过 `modprobe` 命令手动加载：

```
//Intel服务器
[root@mitaka ~]#modprobe -a kvm
[root@mitaka ~]#modprobe -a kvm-intel
//AMD服务器
[root@mitaka ~]#modprobe -a kvm
[root@mitaka ~]#modprobe -a kvm-amd
//加载之后验证内核中是否有KVM相关模块
[root@mitaka ~]# lsmod |grep kvm
kvm_intel          162153  0
kvm                525259  1 kvm_intel
```

在确认主机支持并开启虚拟化功能，同时系统中已经成功加载 KVM 模块后，便可为 Nova 指定使用基于 KVM 的 Hypervisor。要显式告知 Nova 使用 KVM，只需在 Nova 的配置文件 `/etc/nova/nova.conf` 中进行如下设置即可：

```
[root@mitaka ~]#more /etc/nova/nova.conf
[DEFAULT]
compute_driver = libvirt.LibvirtDriver
.....
[libvirt]
virt_type = kvm
.....
```

因为 Nova 默认使用的便是 KVM 虚拟化引擎，如果在 `/etc/nova/nova.conf` 中不显式设置 `virt_type` 值，则 Nova 使用的就是 KVM，对应的如果需要使用其他虚拟化引擎，则需要对 `/etc/nova/nova.conf` 中做对应的修改。在使用 KVM 虚拟化引擎时，其支持的虚拟机镜像有 RAW 格式、QCOW2 (Qemu Copy On Write) 格式、QED (Qemu Enhanced Disk) 格式和 VMDK (VMware virtual machine disk format) 格式。在 KVM 虚拟化引擎下，还有两个与客户机实例相关的重要参数，即客户机后备存储 (Backing Storage) 和客户机 CPU 模式 (CPU Model)。

(1) 客户机后备存储

后备存储，或者称为客户机系统盘，是用来提供客户机操作系统镜像扩展和全部临时存储空间。在虚拟机系统里，后备存储通常是块设备 `/dev/vda`，而在 OpenStack 中，虚拟机的后备存储 `/dev/vda` 可以源自三种不同的选择，分别是 `lvm`、`qcow` 和 `raw`，用户可以通过 `/etc/nova/nova.conf` 中的 `images_type` 进行设置，默认情况下使用的是 `qcow` 格式。

`qcow` 采用 copy-on-write 设计原理进行存储单元的分配，因此，镜像后备存储真正所需的存储空间比起客户机操作系统中所看到的实际存储空间要小得多，如一个 13M 大小的镜像在使用默认的 `qcow` 作为后备存储格式来创建虚拟机时，尽管在客户机中看到的后备存储大小为 1073M (见图 8-36)，但是镜像真正占用的存储空间却仅有十几兆。

```
# fdisk -l
Disk /dev/vda: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders, total 2097152 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
/dev/vda1 *        16065        2088449    1036192+   83   Linux
```

图 8-36 客户机默认块设备

raw 主要用于创建不具备任何类型文件格式的文件，是最原始的磁盘镜像格式。raw 格式的磁盘镜像具有较好的性能，但是在 raw 格式下后备虚拟存储空间会占据全部物理空间，如客户机系统看到的后备存储空间为 1073M，而镜像文件仅有 13M，但是客户机中 1073M 空间会被完全预分配给后备存储使用，尽管后备存储可能只需十几兆的存储空间。raw 格式的优点就是方便转换为不同格式镜像，并在写入时因无须临时分配空间而在性能上优于其他格式，但是 raw 格式的镜像不具备快照、压缩和加密等功能，而且由于 raw 格式镜像占用比实际容量大很多的存储空间，其在迁移过程中也比较耗时。

本地 LVM 卷也可以用于虚拟机后备存储，使用时，只需在 /etc/nova/nova.conf 中进行如下配置即可：

```
[root@mitaka ~]# more /etc/nova/nova.conf
images_type = lvm
images_volume_group = vg_name
```

(2) 客户机 CPU 模式

OpenStack 的 Nova 服务允许用户控制暴露给 KVM 客户机操作系统使用的 CPU 模式。在 Libvirt 中，CPU 被一系列 CPU 模型名称指定，而这些 CPU 模型名被定义在系统文件 /usr/share/libvirt/cpu_map.xml 中，在设置 CPU 模型前，可以先到此文件中核实当前系统支持哪些 CPU 模型名。在使用 KVM 虚拟化引擎时，Nova 的配置文件 /etc/nova/nova.conf 的 [libvirt] 配置段中有两个配置参数来控制客户机 CPU 模式，分别是 cpu_model 和 cpu_mode。其中，cpu_mode 可以设置的值为 none、host-passthrough、host-model 和 custom。

- ❑ host-model：host-model 是 KVM 和 QEMU 虚拟化引擎的客户机默认 CPU 模式。当 cpu_mode 设置为 host_model 时，Libvirt 将采用 /usr/share/libvirt/cpu_map.xml 文件中定义的，并且最匹配物理主机 CPU 的模式作为客户机的 CPU 模式。host_model 模式为客户机提供了最多的功能和性能，而且当客户机在不同 CPU 型号的主机之间进行迁移时，host_model 模式提供了最佳的可靠性和兼容性。
- ❑ host-passthrough：如果设置 cpu_mode 为 host_passthrough，则 Libvirt 将主机 CPU 在不做任何变更的情况下直接暴露给 KVM 虚拟机使用，即让 KVM 虚拟机直接使用物理 CPU。在这种模式下，KVM 虚拟机将获得最佳的性能，并且对于某些需要检查 CPU 底层细节的应用程序而言，让虚拟机直接使用主机 CPU 也是非常重要

的。但是这也会导致一些问题，如当进行虚拟机迁移时，目标主机的 CPU 型号必须与源主机一致，否则虚拟机迁移后不能正常运行或者 Libvirt 不会允许迁移。

- ❑ **custom**：当 `cpu_mode` 设置为 `custom` 时，便可通过 `cpu_model` 参数显示指定用户自定义的 CPU 模型。用户设置的 CPU 模型必须在 `/usr/share/libvirt/cpu_map.xml` 文件中有定义，例如要让客户机使用自定义的 Nehalem CPU，则在 `/etc/nova/nova.conf` 中的 `[libvirt]` 字段应该进行如下配置：

```
[libvirt]
cpu_mode = custom
cpu_model = Nehalem
```

- ❑ **none**：当使用 KVM 或 QEMU 以外的虚拟化引擎时，`cpu_mode` 的值应该设置为 `none`。当 `cpu_mode=none` 时，Libvirt 将不会为 CPU 指定模型，而是由具体的 Hypervisor 选择自己默认的 CPU 模型。

在 Nova 中，针对 Hypervisor 的选取与配置主要分为两个部分，即 Hypervisor 的选项设置和 Libvirt 的选项设置。表 8-7 为 Mitaka 版本中 Nova 计算节点上 `/etc/nova/nova.conf` 中用于配置 Libvirt 的参数。在 `nova.conf` 中，与 Libvirt 相关的参数配置主要位于 `DEFAULT` 和 `libvirt` 配置段，这些参数主要针对虚拟机镜像及其快照、虚拟机 CPU 模式、文件注入和 Hypervisor 选取等进行设置。

表 8-7 Libvirt 配置选项描述

[DEFAULT]	
<code>remove_unused_base_images = True</code>	布尔值，是否删除未使用的 base 镜像
<code>remove_unused_original_minimum_age_seconds = 86400</code>	整数值，删除未使用镜像的最小时间，单位为秒
[libvirt]	
<code>block_migration_flag = VIR_MIGRATE_UNDEFINE_SOURCE, VIR_MIGRATE_PEER2PEER, VIR_MIGRATE_LIVE, VIR_MIGRATE_TUNNELLED, VIR_MIGRATE_NON_SHARED_INC</code>	字符串值，块迁移标志，为避免潜在的配值混淆，此参数将在后续版本中移除
<code>checksum_base_images = False</code>	布尔值，是否进行镜像完整性校验
<code>checksum_interval_seconds = 3600</code>	整数值，镜像完整性校验频率
<code>connection_uri =</code>	字符串，用于覆盖默认的 libvirt URI 值
<code>cpu_mode = None</code>	字符串，客户机 CPU 模式。可选值为 <code>host_model</code> 、 <code>host-passthrough</code> 、 <code>custom</code> 和 <code>none</code> ，如果 <code>virt_type=kvm qemu</code> ，则默认为 <code>host_model</code> ，否则默认为 <code>none</code>
<code>cpu_model = None</code>	字符串，设置 CPU 模型，仅在 <code>cpu_mode=custom</code> 并且 <code>virt_type = kvm qemu</code> 时可用
<code>disk_cachemodes =</code>	列表，设置不同磁盘缓存模式，如 <code>[file=directsync,block=none]</code>

(续)

disk_prefix = None	字符串, 用于覆盖 attach 到虚拟机的设备前缀, 有效值有 sd、xvd、uud 和 vd
images_rbd_ceph_conf =	字符串, ceph 配置文件路径
images_rbd_pool = rbd	字符串, ceph 的 RADOS 存储池名称
images_type = default	字符串, 虚拟机镜像格式
images_volume_group = None	字符串, 当 images_type=lvm 时有效, 设置用于虚拟机镜像的 VG 名称
inject_key = False	布尔值, 是否在启动时诸如 ssh 公钥
inject_partition = -2	整数值, 注入 key 的目标分区序号, -2 表示禁止注入, -1 表示检查, 0 表示没有分区, 大于 0 的数字表示分区号
inject_password = False	布尔值, 在启动时注入镜像用户密码
iscsi_use_multipath = False	布尔值, 是否使用多路径连接 iSCSI 或者 FC 卷
iser_use_multipath = False	布尔值, 是否使用多路径连接 iSER 卷
mem_stats_period_seconds = 10	整数值, 内存使用统计频率, 0 或负数表示禁止内存使用统计
remove_unused_resized_minimum_age_seconds = 3600	整数值, 删除 resize 镜像的最小时间, 单位为秒
rescue_image_id = None	字符串, Rescue 镜像 ID 设置, AMI
rescue_kernel_id = None	字符串, Rescue kernel 镜像 ID 设置, AKI
rescue_ramdisk_id = None	字符串, Rescue 镜像 ID 设置, ARI
rng_dev_path = None	字符串, 提供宿主主机熵源的设备路径, 可用值为 /dev/random 或 /dev/hwrng
snapshot_compression = False	布尔值, 是否压缩镜像快照, 目前只针对 qcow2 有效
snapshot_image_format = None	字符串, 快照镜像格式, 默认与源镜像一致
snapshots_directory = \$instances_path/snapshots	字符串, 上传到 glance 之前存放快照镜像的位置路径
sparse_logical_volumes = False	布尔值, 是否创建稀疏矩阵逻辑卷
sysinfo_serial = auto	字符串, 宿主机传递到客户机虚拟 BIOS 中的“序列号”UUID
uid_maps =	列表, uid 的目标范围, 语法为 guest-uid:host-uid:count, 最大允许值为 5
gid_maps =	列表, gid 的目标范围, 语法为 guest-gid:host-gid:count, 最多允许值为 5
use_usb_tablet = True	布尔值, 是否将真实鼠标与 Windows VMs 中虚拟鼠标同步
use_virtio_for_bridges = True	布尔值, 是否使用 virtio 作为 KVM/QEMU 虚拟机的桥接接口
virt_type = kvm	字符串, Libvirt 域类型, 即 hypervisor 的选取, 可用值为 KVM、QEMU、Xen、LXC、UML 等
volume_clear = zero	字符串, 擦除陈旧卷的方法
volume_clear_size = 0	整数, 旧卷擦除大小, 单位为 MB, 0 表示全部擦除
wait_soft_reboot_seconds = 120	整数, 发起软重启命令后等待实例 shutdown 的时间, 单位为秒, 如果超出此时间实例没有 shutdown, 则强制进行硬重启

表 8-8 为 Mitaka 版本中 Nova 计算节点上 `/etc/nova/nova.conf` 中关于 Hypervisor 的配置选项。在 `nova.conf` 中，关于 Hypervisor 的配置选项在 DEFAULT 配置段进行，主要涉及 Hypervisor 所使用的镜像、临时存储、格式化和物理 CPU 绑定等配置。

表 8-8 Hypervisor 配置选项描述

[DEFAULT]	
default_ephemeral_format = None	字符串，临时存储被格式化时的默认格式，可能的值为 ext2、ext3、ext4、xfs、ntfs (only for Windows guests)
force_raw_images = True	布尔值，是否强制转换后端镜像 (backing images) 为 raw 格式，如果为 true，则 backing images 被转换为 raw 格式，如果为 false，则 backing images 不会被转换
preallocate_images = none	字符串，要使用的镜像预分配模式，可能的值为 none 和 space，none 表示不进行预分配，space 表示在实例启动时预分配
timeout_nbd = 10	整数值，等待 NDB 驱动启动的时间，单位为秒
use_cow_images = True	布尔值，是否启用 copy-on-write (cow) 镜像，QEMU/KVM 允许使用 qcow2 作为后端文件，如果被禁用，则后端文件不会被使用
vcpu_pin_set = None	字符串，定义那些物理 CPUs (pCPUs) 可以被实例的虚拟 CPUs (vCPUs) 使用，参数值由可用物理 CPU 编号组成的逗号列表，如果禁止使用某颗 CPU，则可以在其前面加上 “^” 符号，如 <code>vcpu_pin_set = “4-12,^8,15”</code> 表示 4-12 号和 15 号 CPU 均可用，但是 8 号 CPU 不可用
virt_mkfs = []	多值参数，用于临时设备的 mkfs 命令名称，格式为 <code><os_type>=<mkfs command></code>

8.5 Nova 主机策略

在 Nova 的各个服务组件中，Nova-scheduler 是个非常关键的组件，其主要作用便是为 Nova 的主机选取提供智能决策功能。当 Nova 客户端发起创建实例请求时，Nova-API 会将请求转发到 Nova-scheduler，由 Nova-scheduler 在运行 Nova-compute 的计算节点中选取用于创建符合请求虚拟机资源条件的宿主机。Nova-scheduler 对宿主机的选取分为两个步骤：第一步从计算节点集群中选取符合请求虚拟机资源条件的全部节点，这一过程称为过滤 (Filter)；第二步从符合创建请求虚拟机的计算节点中选取唯一最佳的计算节点，作为本次请求虚拟机的宿主机，这一过程称为加权 (Weigh)。Nova-scheduler 的过滤加权过程如图 8-37 所示，在经过第一步的过滤之后，host2 和 host4 被排除，即只有 host1、host3、host5 和 host6 符合创建请求虚拟机的条件，这四个主机经过第二步的加权后，得到一个主机权重排序列表，此处的 host5 主机便是创建请求虚拟机的最佳宿主机，即本次虚拟机创建请求会被指派到 host5 主机并最终在 host5 上创建虚拟机。

8.5.1 Nova scheduler 主机过滤

在 Nova-scheduler 的过滤阶段，Nova 的 Filter Scheduler 根据配置文件 `nova.conf` 中设

置的策略集合 (Filters Set) 对 OpenStack 集群中全部计算节点进行过滤迭代。在 nova.conf 中, 通过三个参数来配置 Nova-scheduler, 即 scheduler_driver、scheduler_available_filters 和 scheduler_default_filters, 这三个参数的默认值如下:

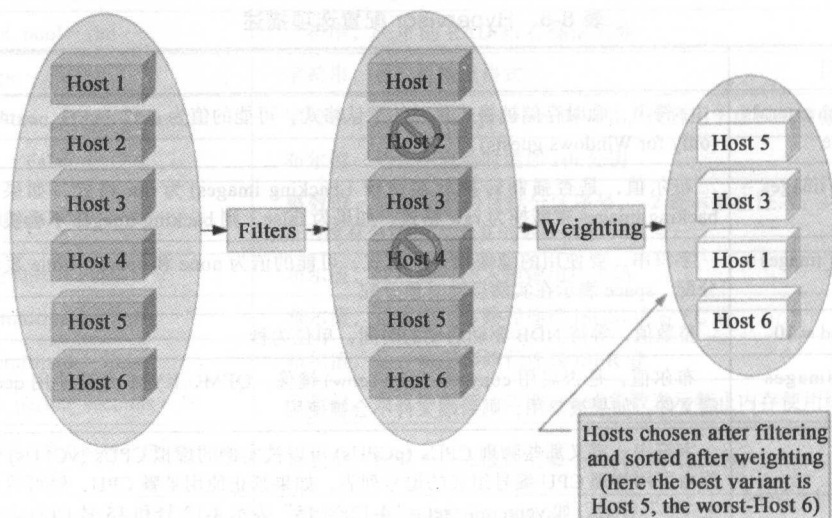


图 8-37 Nova-scheduler 对计算节点进行过滤与加权

```

//设置使用的scheduler驱动, 这里为FilterScheduler
scheduler_driver=nova.scheduler.filter_scheduler.FilterScheduler
//配置FilterScheduler可用的 filter, 默认所有filters都可用
scheduler_available_filters = nova.scheduler.filters.all_filters
//显式指定FilterScheduler在过滤时使用的 filters
scheduler_default_filters = RetryFilter, AvailabilityZoneFilter,\
    RamFilter, DiskFilter, ComputeFilter, ComputeCapabilitiesFilter,\
    ImagePropertiesFilter, ServerGroupAntiAffinityFilter,\
    ServerGroupAffinityFilter
  
```

在 Nova 的配置文件 /etc/nova/nova.conf 中, 配置参数 scheduler_driver 用于设置 Nova-scheduler 使用的 Scheduler, FilterScheduler(nova.scheduler.filter_scheduler.FilterScheduler) 是 Nova-scheduler 默认使用的 Scheduler。此外, Nova-scheduler 允许使用第三方的 Scheduler, 用户只需设置 scheduler_driver 即可。在使用默认的 FilterScheduler 时, Nova-scheduler 首先通过过滤器筛选出满足条件的计算节点, 再通过加权计算选择最优的计算节点以在其上创建实例。

在 nova.conf 中, 配置参数 scheduler_available_filters 用于指定 Scheduler 可以使用的过滤器, 默认情况下 Nova 自带的全部过滤器均可被使用 (nova.scheduler.filters.all_filters)。可用过滤器参数可以重复指定, 如用户自己实现了一个过滤器 myfilter.WarriorFilter, 然后用户既想使用 Nova 自带的过滤器又想使用自建的过滤器, 则可以将可用过滤器参数 scheduler_available_filters 在 nova.conf 中进行如下重复配置:

```
scheduler_available_filters = nova.scheduler.filters.all_filters
scheduler_available_filters = myfilter.WarriorFilter
```

nova.conf 中的另外一个配置参数 scheduler_default_filters 用于指定 Nova-scheduler 服务使用的过滤器列表。在 Nova-scheduler 进行主机过滤时，主机会被此参数指定的列表过滤器顺序过滤一遍。Nova-scheduler 默认使用的过滤器依次为 RetryFilter、AvailabilityZoneFilter、RamFilter、ComputeFilter、ComputeCapabilitiesFilter、ImagePropertiesFilter、ServerGroupAntiAffinityFilter 和 ServerGroupAffinityFilter，在主机进入每个过滤器时，如果主机条件满足请求中的实例需求，则返回 True，否则返回 False。当返回 False 后，该主机的过滤流也就结束，不再进入后续过滤器继续过滤。当然，Nova-scheduler 内建自带的过滤器不局限于上述 8 个，下面将分别对 Nova-scheduler 的主要内建过滤器进行介绍。

(1) AggregateCoreFilter

以 Aggregate 为前缀的过滤器通常只在创建了 Host Aggregate 的情况下才会使用到。AggregateCoreFilter 表示通过主机 CPU 核数来过滤每个 Aggregate 中的主机。主机可用 CPU 核心数通过每个主机集配置文件中的 cpu_allocation_ratio 值（默认为 16）来计算。如果主机 CPU 核心数满足请求创建实例的 vCPUS 需求，则返回 True。如果当前主机属于多个 Host Aggregate，并且有多个 cpu_allocation_ratio 值，则使用 cpu_allocation_ratio 的最小值。cpu_allocation_ratio 参数主要用于设置主机 CPU 的 overcommit，如 cpu_allocation_ratio=16，而主机 vCPUs 为 8，则调度时 Scheduler 认为主机可用 vCPU 为 128。

(2) AggregateDiskFilter

AggregateDiskFilter 表示通过主机磁盘使用率来过滤每个 Aggregate 中的主机。主机磁盘使用率会根据每个主机集配置文件中的 disk_allocation_ratio（默认值为 1，即按实际磁盘容量来调度）参数来计算，如果 OpenStack 集群中没有创建主机集，则 disk_allocation_ratio 便会成为全局性的参数。如果主机集中主机上的可用磁盘容量（单位为 GB）满足请求虚拟机中的磁盘需求（镜像系统盘大小和临时存储大小），则返回 True。

(3) AggregateImagePropertiesIsolation

AggregateImagePropertiesIsolation 表示根据实例创建请求中的镜像属性来过滤主机集中的主机，通常用于将基于特定镜像的实例创建到与镜像匹配的特定主机上。镜像属性与主机匹配与否根据如下规则来判断：如果当前主机属于某个主机集，并且此主机集定义了相应的元数据来匹配某个镜像的属性，则认为该主机与镜像属性匹配，并且该主机将成为启动这个匹配镜像的候选主机；如果当前主机不属于任何主机集，则该主机可以启动任何镜像。例如对于主机集 HA1，有如下的主机成员和元数据信息：

```
#nova aggregate-details HA1
```

Id	Name	Availability Zone	Hosts	Metadata
1	MyWinAgg	None	'host-1'	'os=windows'

主机集 HA1 有主机成员 host-1 和元数据 “os=windows”，而 Glance 中的镜像 windows2012 也具有如下的元数据信息：

```
# glance image-show windows2012
```

Property	Value
Property 'os'	windows
checksum	f8a2eeee2dc65b3d9b6e63678955bd83
container_format	ami
created_at	2013-11-14T13:24:25

由于镜像 Windows2012 具有与主机集 HA1 相同的元数据 “os=windows”，所以，当 Nova 客户端发起基于镜像 Windows2012 的虚拟机创建请求时，此例中的虚拟机将会创建在 host-1 主机上。

(4) AggregateInstanceExtraSpecsFilter

AggregateInstanceExtraSpecsFilter 与 AggregateImagePropertiesIsolation 过滤器的功能类似，只是 AggregateInstanceExtraSpecsFilter 匹配的是 Nova 中 Flavor 与主机集的元数据，而不是镜像属性与主机集的元数据。在使用 AggregateInstanceExtraSpecsFilter 过滤器时，通常需要为 Flavor 的属性 aggregate_instance_extra_specs 设置一个属性值。过滤器将会根据主机集的元数据与实例创建请求中 Flavor 的 aggregate_instance_extra_specs 属性值来判断主机是否符合要求。关于此过滤器的具体使用可以参考 8.3.4 节。

(5) AggregateIoOpsFilter

AggregateIoOpsFilter 表示根据主机的 IO 负载来过滤每个主机集中的主机。主机 IO 负载根据参数 max_io_ops_per_host 值（默认为 8）来决定，此过滤器会将 IO 负载过高的主机过滤掉。

(6) AggregateNumInstancesFilter

AggregateNumInstancesFilter 表示根据主机已有实例数目来过滤每个主机集中的主机，过滤标准为主机集中允许的最大主机实例数目参数 max_instances_per_host（默认 50）值，当前运行实例数目超过此参数值的主机将会被过滤掉。

(7) AggregateMultiTenancyIsolation

AggregateMultiTenancyIsolation 用于将特定某个租户或多个租户的实例创建到特定的主机集或可用域中。如果主机属于某个具有元数据且 key 为 filter_tenant_id 的主机集，则对应 filter_tenant_id 值的某个租户或多个租户发起的创建实例请求将会在该主机上创建实例。而如果主机不属于元数据 key 为 filter_tenant_id 的主机集，则所有组合都可以在该主机上创建实例。AggregateMultiTenancyIsolation 过滤器并不会将具有元数据 Key 为 filter_tenant_id 的主机集与其他租户隔离，任何租户仍然可以在指定的主机集上创建实例。

(8) AggregateRamFilter

AggregateRamFilter 表示通过主机可用内存 RAM 来过滤每个主机集中的主机，主机的可用内存根据主机集配置文件中 `ram_allocation_ratio` 值（默认为 1.5）来计算。

(9) AggregateTypeAffinityFilter

AggregateTypeAffinityFilter 用于通过主机集的 `instance_type` 键来过滤主机，如果主机集没有设置 Key 为 `instance_type` 的元数据，或者将主机集中 key 为 `instance_type` 的元数据值设为实例创建请求中所包含的 `instance_type` 值，则请求中的 `instance_type` 的元数据值可以是单个字符串或者逗号分开的多个字符串，如 `m1.nano` 或 `m1.nano,m1.small`。

(10) AllHostsFilter

AllHostsFilter 不会发生任何过滤操作，所有正常运行的计算节点都会通过该过滤器。

(11) AvailabilityZoneFilter

AvailabilityZoneFilter 是可用域 AZ 过滤器，当在请求中指定创建实例的 AZ 时，必须在 Scheduler 中使用该过滤器。

(12) ComputeCapabilitiesFilter

ComputeCapabilitiesFilter 与 AggregateInstanceExtraSpecsFilter 类似，也是通过设置 Nova 中主机类型的元数据来对主机进行过滤，不过 ComputeCapabilitiesFilter 过滤器中与 Flavor 的元数据匹配的对象不是主机集中的 metadata，而是主机的 Capabilities。Capabilities 可以是主机的 Architecture、Cores、Features、Model 和 Vendor 等，在使用 ComputeCapabilitiesFilter 过滤器时，Flavors 的 extra specs 格式为“Capabilities:key:value”形式，“Capabilities”表示命名空间，“key:value”表示 extra specs 键值对。如 extra specs 为“Capabilities:architecture:x86”的 Flavor 在创建实例时，ComputeCapabilitiesFilter 只会允许 X86 架构的主机通过，如果命名空间不是“Capabilities”，则 Flavors 的 extra specs 中命名空间字段会被忽略而仅提取“key:value”部分。在使用 ComputeCapabilitiesFilter 过滤器时，强烈建议不要省略命名空间（extra specs 中第一个冒号前的字符串），否则将会与 AggregateInstanceExtraSpecsFilter 过滤器冲突（两个过滤器同时启用的情况下），同时建议将命名空间字符串设为 Capabilities。

(13) ComputeFilter

ComputeFilter 意味着正常运行 nova-compute 服务的计算节点才能被 Nova-scheduler 调度，通常而言，ComputeFilter 是必须的过滤器。

(14) CoreFilter

CoreFilter 与 AggregateCoreFilter 过滤器类似，按主机 CPU 核数来过滤主机。如果未启用此过滤器，则 Scheduler 调度给实例的 CPU 核数会超出物理主机的实际 CPU 核数，即运行在实例上的虚拟 CPU 核数之和可以超出物理主机 CPU 核数。通过启用 CoreFilter 过滤器，并且设置超额使用比率参数 `cpu_allocation_ratio`，限定实例可以超额使用的虚拟 CPU 核数。例如在启用 CoreFilter 的情况下，`cpu_allocation_ratio` 设置为 8，主机物理 CPU 核

数为 8，则该主机可供实例使用的全部 vCPU 核数为 64。如果希望实例 vCPU 与主机物理 CPU 为 1:1 的分配使用关系，则可以设置 `cpu_allocation_ratio` 为 1。

(15) DifferentHostFilter

DifferentHostFilter 将请求实例创建在与请求所指定的实例主机不同的主机上，即指定新创建的实例不能位于某个或多个实例的宿主机上。如 Nova 中已经有两个实例，并且用户希望新创建的实例不能与这两个已有实例的宿主机相同，则可以启用 DifferentHostFilter 过滤器，并使用如下语句创建实例：

```
#nova boot --image cedef40a-ed67-4d10-800e-17455edce175 --flavor 1 \
--hint different_host=a0cf03a5-d921-4877-bb5c-86d26cf818e1 \
--hint different_host=8c19174f-4220-44f0-824a-cd1eeef10287 server-1
```

(16) DiskFilter

DiskFilter 与 AggregateDiskFilter 过滤器类似，即仅调度磁盘空间满足请求实例的 root 和临时存储空间的主机。在 Nova 中，主机磁盘可以超额使用，可以通过配置磁盘超额使用比率参数 `disk_allocation_ratio` 来限定调度器可见的最大虚拟主机磁盘容量。在配置文件 `nova.conf` 中，`disk_allocation_ratio` 的默认值为 1，即不允许主机磁盘空间被超额分配，在调度时以主机的实际可用磁盘容量为准。此外，需要注意的是，Scheduler 提取的主机磁盘空间并不是 hypervisor 统计结果中 `free_disk_gb` 的值，而是 `disk_available_least` 的值。

```
[root@mitaka ~]# nova hypervisor-stats
```

Property	Value
count	1
current_workload	0
disk_available_least	14
free_disk_gb	25
free_ram_mb	6784
local_gb	26
local_gb_used	1
memory_mb	7808
memory_mb_used	1024
running_vms	1
vcpus	2
vcpus_used	1

在上述 hypervisor 统计结果中，Scheduler 用于调度的主机可以存储空间值为 `disk_available_least=14GB`，而不是 `free_disk_gb=25GB`。此外，如果想要实例可以超额使用主机存储空间，只需将参数 `disk_allocation_ratio` 设置为大于 1 即可。

(17) RamFilter

RamFilter 过滤器将根据主机是否有足够的可用 RAM 来进行过滤。如果未启用 RamFilter 过滤器，则 Scheduler 会从主机上提取超额的 RAM，即分配给实例的 RAM 之和会大

于主机物理 RAM 容量。在启用 RamFilter 后，可以通过配置 RAM 超额使用比率参数 `ram_allocation_ratio` 的值来限定该主机可用的最大虚拟内存值，`ram_allocation_ratio` 的默认值为 1.5，因此如果当前主机可用物理 RAM 为 10GB，则 Scheduler 提取到的主机 RAM 为 15GB。

(18) JsonFilter

JsonFilter 过滤器允许用户通过 Json 格式的 Scheduler 调度提示 (hint) 来自定义主机调度。JsonFilter 过滤器允许的操作符有 =、>、<、in、<=、>=、not、or 和 and，允许使用的变量有 `$free_ram_mb`、`$free_disk_mb`、`$total_usable_ram_mb`、`$vcpus_total` 和 `$vcpus_used`。如希望将实例创建在主机可用 RAM 大于等于 2GB 的主机上，则在启用 JsonFilter 过滤器的情况下，使用如下实例创建语句：

```
[root@mitaka ~]#nova boot --image 827d564a-e636-4fc4-a376-d36f7ebe1747 --flavor 1 --hint query='[">=", "$free_ram_mb", 2048]' instance-1
```

(19) RetryFilter

RetryFilter 用于将之前已经调度过的主机过滤掉。此过滤器只有在主机过滤重复参数 `scheduler_max_attempts` 值大于 1 的情况下才有效。如 A、B、C 三台主机在某次过滤后均符合标准，并且 A 主机因为权重最优而被选为创建实例的主机，但是由于某些原因实例创建失败，同时参数 `scheduler_max_attempts` 值大于 1，因此 Scheduler 将进行再次调度，为了避免再次失败，当启用 RetryFilter 过滤器时，A 主机将不再参与调度。为了避免反复过滤失败主机，RetryFilter 通常是默认过滤器列表中的第一个过滤器。

(20) SameHostFilter

SameHostFilter 的功能与 DifferentHostFilter 相反，即将请求实例创建在指定实例的宿主机上，使得要创建的实例与一个或多个已经存在的实例位于同一主机上。要使用此过滤器，客户端在创建实例时需要通过 hint 传入 `same_host` 键，键值为特定实例的 UUID。如已经存在两个实例，要将新创建的实例位于这两个实例所在的宿主机，则可以通过如下命令创建实例：

```
[root@mitaka ~]#nova boot --image cedef40a-ed67-4d10-800e-17455edce175 --flavor 1 --hint same_host=a0cf03a5-d921-4877-bb5c-86d26cf818e1 --hint same_host=8c19174f-4220-44f0-824a-cd1eeef10287 host-2
```

(21) ServerGroupAffinityFilter

ServerGroupAffinityFilter 与 GroupAffinityFilter 功能相同，而 GroupAffinityFilter 即将被淘汰，因此建议使用 ServerGroupAffinityFilter。过滤器 ServerGroupAffinityFilter 的作用是将请求实例调度到指定的 Server Group 中（此处的 Server 表示虚拟机），即将实例尽量创建到同一台主机上。要使用此过滤器，用户必须使用“affinity”策略预先创建一个服务器组（实例组），并在创建实例时通过 hint 参数的 `group` 键值对指定该实例属于此服务器组，`group` 键的值为服务器组的 UUID。在启用 ServerGroupAffinityFilter 的情况下，客户端使用方法如下：

```
[root@mitaka ~]#nova server-group-create --policy affinity group-1
[root@mitaka ~]#nova boot --image cedef40a-ed67-4d10-800e-17455edce175\
--flavor 1 --hint group= 4a4c8c1b-8358-4f5b-ad20-ed553dbd5683 host-1
[root@mitaka ~]#nova boot --image cedef40a-ed67-4d10-800e-17455edce175\
--flavor 1 --hint group= 4a4c8c1b-8358-4f5b-ad20-ed553dbd5683 host-2
```

这里，实例 host-1 与 host-2 属于相同的服务器组 group-1，策略为 affinity，因此两个实例将位于相同的主机上。

(22) ServerGroupAntiAffinityFilter

ServerGroupAntiAffinityFilter 与 ServerGroupAffinityFilter 正好相反，此外，以前的过滤器 GroupAntiAffinityFilter 即将被丢弃，因此建议使用 ServerGroupAntiAffinityFilter 过滤器。ServerGroupAntiAffinityFilter 表示将实例尽量不要创建在同一台主机上。要启用此过滤器，用户需要使用“anti-affinity”策略预先创建服务器组，并在创建实例时通过 hint 参数的 group 键值对指定该实例属于此服务器组，group 键的值为服务器组的 UUID。在启用 ServerGroupAntiAffinityFilter 的情况下，客户端使用方法如下：

```
[root@mitaka ~]#nova server-group-create --policy anti-affinity group-1
[root@mitaka ~]#nova boot --image cedef40a-ed67-4d10-800e-17455edce175\
--flavor 1 --hint group=9076aa90-7c2a-4095-aad1-bd776de3d069 host-1
[root@mitaka ~]#nova boot --image cedef40a-ed67-4d10-800e-17455edce175\
--flavor 1 --hint group=9076aa90-7c2a-4095-aad1-bd776de3d069 host-2
```

这里，实例 host-1 与 host-2 属于相同的服务器组 group-1，策略为 anti-affinity，因此两个实例将位于不同的主机上。

(23) SimpleCIDRAffinityFilter

SimpleCIDRAffinityFilter 是一个基于网络的过滤器，主要用于根据创建实例时指定的 IP 地址范围来过滤主机。要使用此过滤器，用户在创建实例时必须指定两个 hint 参数，hint 参数的两个 Key 分别为 build_near_host_ip 和 cidr，第一个键的值为有效的 IP 地址，第二个键的值为 CIDR 格式的掩码。SimpleCIDRAffinityFilter 的使用方法如下：

```
[root@mitaka ~]#nova boot --image cedef40a-ed67-4d10-800e-17455edce175\
--flavor 1 --hint build_near_host_ip=192.168.1.1 --hint cidr=/24 host-1
```

8.5.2 Nova scheduler 主机加权

在经过 SchedulerFilter 中一系列的过滤器过滤之后，Nova-scheduler 会选出符合请求实例资源的主机，但是如果有多台主机同时符合要求，则经过 SchedulerFilter 之后，仍然会得到一个主机列表。为了得到最终的实例创建主机，Nova-scheduler 会给过滤后的主机列表中的计算节点进行“评分”，分数最高的主机将成为最终的实例创建主机，这个“评分”的过程即是所谓的权重计算（Weighting）。权重计算就是给有效主机列表中的主机进行权重赋值并选出最优权重值主机的过程。每个计算节点都可以通过不同的维度参数来进行评估，如内存、磁盘、CPU 和 IO 负载，而这些不同维度的参数被称为一个权重（Weigher）。由于

各个权重的单位不同，如内存单位为 MB，磁盘单位为 GB，CPU 单位为核数等，因此不能对这些权重进行简单的线性相加来获取计算节点最终的权重值。同时，不同的权重对用户而言可能重要程度并不一样（如 A 用户认为内存比 CPU 重要，而 B 用户认为 CPU 比内存更重要）。为了解决这一问题，需要预先为每个权重定义关联的权重因子（multiplier），如果认为此权重优于其他权重，则可以为其设置较大的权重因子，nova.conf 配置文件中与权重相关的默认参数配置如下：

```
[DEFAULT]
scheduler_host_subset_size = 1           //按照排序后主机列表仅选取一台主机
scheduler_weight_classes = nova.scheduler.weights.all_weighters
ram_weight_multiplier = 1.0              //权重因子
io_ops_weight_multiplier = 2.0           //权重因子
soft_affinity_weight_multiplier = 1.0    //权重因子
soft_anti_affinity_weight_multiplier = 1.0 //权重因子
[metrics]
weight_multiplier = 1.0
weight_setting = name1=1.0, name2=-1.0
required = false
weight_of_unavailable = -10000.0
```

最后，各个权重与其权重因子的乘积之和便是最终的计算节点权重值。需要指出的是，各个权重在参与计算之前会事先进行标准化（normalize），因此，节点最终的权重计算公式如下：

$$\text{weight} = w1_multiplier * \text{norm}(w1) + w2_multiplier * \text{norm}(w2) + \dots$$

其中，w1_multiplier 表示权重 w1 的权重因子，weight 表示节点最终的权重和，norm(w1) 表示对权重值 w1 进行标准化，这里 w1 通常称为原始权重值，即通过 Weigher 获取的最初值。假设 w1 为 RAMWeigher，则 RAMWeigher 获取的主机可用 RAM 值即是 w1 的原始权重值，原始权重值的标准化公式如下：

$$\text{ratio_weight} = (\text{raw_weight} - \text{min_weight}) / (\text{max_weight} - \text{min_weight})$$

其中，raw_weight 表示某个主机针对特定权重的原始权重值，而 min_weight 表示被计算权重的所有主机中，针对特定权重的最小原始权重值，max_weight 则是最大原始权重值。现假设有 host1、host2、host3、host4、host5 和 host6 六台主机需要进行权重排序，并且有 3 个权重，分别为 W1、W2 和 W3，而 3 个权重对应的权重因子分别为 1、1、2。六台主机经过 W1、W2 和 W3 计算后得到的原始权重值分别如下：

主机名称	HOST1	HOST2	HOST3	HOST4	HOST5	HOST6
W1 原始权重	110	20	10	80	50	90
W2 原始权重	4	2	1	11	6	8
W3 原始权重	5	25	10	15	5	20

根据标准化计算公式，六台主机针对 W1、W2 和 W3 原始权重值进行标准化计算后的结果分别如下：

主机名称	HOST1	HOST2	HOST3	HOST4	HOST5	HOST6
norm(W1)	1	0.1	0	0.7	0.4	0.8
norm(W2)	0.3	0.1	0	1	0.5	0.7
norm(W3)	0	1	0.25	0.5	0	0.75

由于 W1、W2 和 W3 的权重因子分别为 1、1、2，因此根据节点最终的权重计算公式，六台主机节点的最终权重值分别如下：

主机名称	HOST1	HOST2	HOST3	HOST4	HOST5	HOST6
weight	1.3	2.2	0.5	2.7	0.9	3.0

对主机列表中的主机依据各自权重值进行降序排列后，得到的排序结果为 host6、host4、host2、host1、host5、host3，因此根据权重排序后的最优节点应该是 host6 节点。如果各个权重的权重因子为负数，则 host3 节点为最优节点。关于节点加权排序的过程描述，OpenStack 的官方网站给出了形象图示（如图 8-38）。在 FilterScheduler 调度过程的末尾，通过过滤的主机列表被根据各自的权重值进行排序，此处权重值最小的节点被认为最优，并作为创建实例的最终节点，而在图 8-38 中，主机 host4 权重值最小，因此排在最前面，成为此次调度的最优主机。

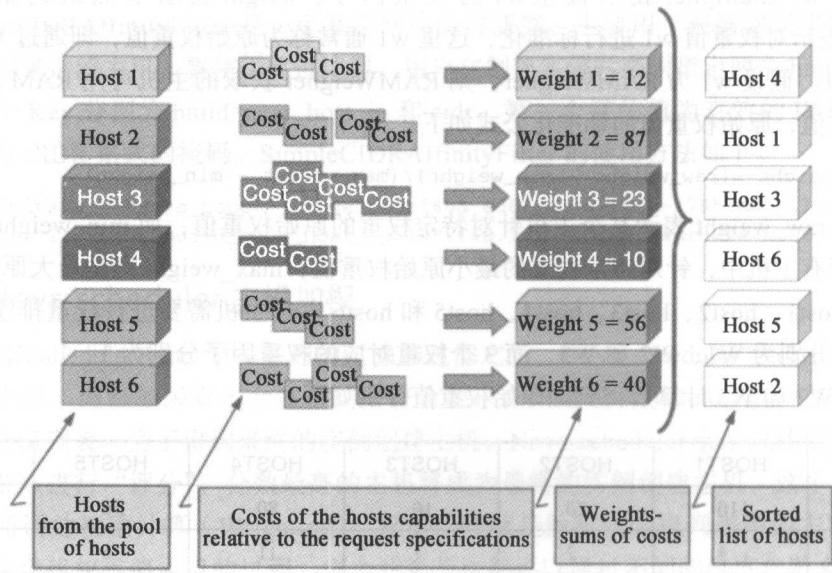


图 8-38 Nova-scheduler 节点加权计算过程

在实际使用中, FilterScheduler 根据主机配置文件 nova.conf 中的 scheduler_weight_classes 配置参数对主机进行不同的加权计算, 此参数的默认配置如下:

```
scheduler_weight_classes=nova.scheduler.weights.all_weighters
```

即默认使用全部 Nova 自带的 Weigher, 用户也可以自定义 scheduler_weight_classes 参数值, 其值是字符串列表, 列表中每个元素代表一个 Weigher 的路径名称。此参数的默认值 nova.scheduler.weights.all_weighters 包含了 Nova 自带的常用 Weigher, 它们是 RAMWeigher、DiskWeigher、MetricsWeigher、IoOpsWeigher、ServerGroupSoftAffinityWeigher 和 ServerGroupSoftAntiAffinityWeigher, 如下是对各个 Weigher 的功能使用介绍。

- ❑ RAMWeigher: 表示根据计算节点的可用 RAM 进行权重计算, 剩余可用 RAM 最大的节点具有最优权重值。如果权重因子是负数, 则剩余可用 RAM 最小的节点具有最优权重值。
- ❑ DiskWeigher: 表示根据计算节点的可用磁盘空间进行权重计算, 剩余可用磁盘空间最大的节点具有最优权重值。如果权重因子是负数, 则剩余可用磁盘空间最小的节点具有最优权重值。
- ❑ MetricsWeigher: 表示根据计算节点的各自定义指标进行权重计算, 要成为计算权重的自定义指标参数需要在配置文件中指定, 设置语法如下:

```
metrics_weight_setting = name1=1.0, name2=-1.0
```

- ❑ IoOpsWeigher: 表示根据计算节点的工作负载来计算权重值, 默认选择负载较轻的计算节点作为最优节点, 如果权重因子是正数, 则会认为负载较大的节点最优, 即此时效果刚好也默认情况相反。
- ❑ ServerGroupSoftAffinityWeigher: 表示根据主机上运行在服务器组 (Server Group) 中的实例数目来计算节点权重值, 实例将会创建到权重值最大的主机上。在使用该 Weigher 时, 只有对应的权重因子设为正数才有效, 因为负权重因子意味着不要将实例创建在同一台主机上, 正好与目标结果相反。
- ❑ ServerGroupSoftAntiAffinityWeigher: 表示根据主机上运行在服务器组 (Server Group) 中的实例数目来计算节点权重值, 其权重值为负数, 实例将会创建到权重值最大的主机上 (意味着服务器组中的实例数目最少)。在使用该 Weigher 时, 只有对应的权重因子设为正数才有效, 因为负权重因子意味着将实例尽量创建在同一台主机上, 正好与目标结果相反。

下面来验证 Nova-scheduler 的 Filter 和 Weigh 过程。实验环境中 OpenStack 为 Mitaka 版本, 部署有两个计算节点, 分别为 compute1 和 compute2, 两个计算节点仅运行 Nova-compute 服务, Nova-scheduler 服务部署在控制节点上, 并且 scheduler 使用默认配置, 即关于过滤器的参数配置如下:

```
scheduler_available_filters = nova.scheduler.filters.all_filters
```

```
scheduler_default_filters = RetryFilter, AvailabilityZoneFilter,\
    RamFilter, DiskFilter, ComputeFilter, ComputeCapabilitiesFilter,\
    ImagePropertiesFilter, ServerGroupAntiAffinityFilter,\
    ServerGroupAffinityFilter
```

现在控制节点上创建实例，然后跟踪 scheduler 的日志 /var/log/nova/nova-scheduler.log，注意需要在 nova.conf 中设置 debug=true，在控制节点创建实例的命令如下：

```
[root@controller1 ~]# nova boot --flavor ml.tiny --image\ cirros-0.3.4-x86_64
--nic net-id=edc57f9e-a98c-4ef4-94c3-d1fb5d4d9054\ --security-group default
--key-name admin-key admin-instance3
```

在实例创建过程中，nova-scheduler.log 日志文件中与 Filter 和 Weigh 相关的日志截取片段如下：

```
.....
- - -] Starting with 2 host(s) get_filtered_objects /usr/lib/python2.7/site-
packages/nova/filters.py:70
- - -] Filter RetryFilter returned 2 host(s) get_filtered_objects /usr/lib/
python2.7/site-packages/nova/filters.py:104
- - -] Filter AvailabilityZoneFilter returned 2 host(s) get_filtered_objects /
usr/lib/python2.7/site-packages/nova/filters.py:104
- - -] Filter RamFilter returned 2 host(s) get_filtered_objects /usr/lib/
python2.7/site-packages/nova/filters.py:104
- - -] Filter DiskFilter returned 2 host(s) get_filtered_objects /usr/lib/
python2.7/site-packages/nova/filters.py:104
- - -] Filter ComputeFilter returned 2 host(s) get_filtered_objects /usr/lib/
python2.7/site-packages/nova/filters.py:104
- - -] Filter ComputeCapabilitiesFilter returned 2 host(s) get_filtered_objects
/usr/lib/python2.7/site-packages/nova/filters.py:104
- - -] Filter ImagePropertiesFilter returned 2 host(s) get_filtered_objects /usr/
lib/python2.7/site-packages/nova/filters.py:104
- - -] Filter ServerGroupAntiAffinityFilter returned 2 host(s) get_filtered_
objects /usr/lib/python2.7/site-packages/nova/filters.py:104
- - -] Filter ServerGroupAffinityFilter returned 2 host(s) get_filtered_objects
/usr/lib/python2.7/site-packages/nova/filters.py:104
- - -] Filtered [(compute1, compute1) ram: 800MB disk: 11264MB io_ops: 0
instances: 1, (compute2, compute2) ram: 800MB disk: 11264MB io_ops: 0
instances: 1]
- - -] Weighed [WeighedHost [host: (compute1, compute1) ram: 800MB disk: 11264MB
io_ops: 0 instances: 1, weight: 2.0], WeighedHost [host: (compute2, compute2)
ram: 800MB disk: 11264MB io_ops: 0 instances: 1, weight: 2.0]
- - -] Selected host: WeighedHost [host: (compute1, compute1) ram: 800MB disk:
11264MB io_ops: 0 instances: 1, weight: 2.0]
.....
```

从 nova-scheduler.log 的日志中可以看到，调度器 Scheduler 在 Filter 阶段使用了配置参数 scheduler_default_filters 指定的全部 Filter，并且两个计算节点全部通过了 Filter 的筛选（每个 Filter 都返回两个可用主机）。在这里，两个计算节点的资源使用情况几乎一致，可

用 RAM 为 800MB，可用磁盘空间为 11264MB，io_ops 为 0，两个节点上均有一个实例在运行。Weigh 阶段，通过两个节点的权重值计算，compute1 和 compute2 节点的最终权重值均为 2，而 Scheduler 最后选择了 compute1 节点，即实例 admin-instance3 的宿主机应该为 compute1。可以通过 nova show 命令进行验证，如下：

```
[root@controller1 ~]# nova show admin-instance3
```

Property	Value
OS-DCF:diskConfig	MANUAL
OS-EXT-AZ:availability_zone	nova
OS-EXT-SRV-ATTR:host	compute1
OS-EXT-SRV-ATTR:hostname	admin-instance3
OS-EXT-SRV-ATTR:hypervisor_hostname	compute1
OS-EXT-SRV-ATTR:instance_name	instance-0000000d

从上述输出中可以看出，admin-instance3 实例的 host 为 compute1，instance_name 为 instance-0000000d。再次通过检查 compute1 这个 hypervisor 主机上有多少个实例来进行反向验证，如下：

```
[root@controller1 ~]# nova hypervisor-servers compute1
```

ID	Name	Hypervisor ID
Hypervisor Hostname		
7fba6680-d5d7-4e64-8e85-143e362d1998	instance-0000000a	1
compute1		
2704539e-f955-4972-bcd0-fb773f96ee61	instance-0000000d	1
compute1		

可以看到，hypervisor 为 compute1 的节点上有两个实例，其中之一便是本例创建的实例 admin-instance3，其对应的 instance_name 为 instance-0000000d。上述结果表明，实例 admin-instance3 的创建请求确实被 Scheduler 调度到 compute1 计算节点上执行。

8.5.3 Nova scheduler 配置选项

为了让用户可以灵活配置各个过滤器 (Filter) 和权重 (Weigher)，Nova 在配置文件 /etc/nova/nova.conf 中设置了很多与 Nova-scheduler 相关的配置参数。在这些配置参数中，部分参数使得用户可以自定义 Filter 和 Weigher，而部分参数用于控制具体的 Filter 和 Weigher 的行为，具体与 Nova-scheduler 相关的参数及其使用介绍如表 8-9 所示。

表 8-9 Nova-scheduler 相关配置参数描述

[DEFAULT]	
aggregate_image_properties_isolation_namespace=None	字符串值，用户可以对镜像和主机进行相应的配置，使得基于特定镜像的实例创建请求被调度到特定主机集中的主机上。这一过程是通过主机集中的元数据值来设置的，而主机集元数据的起始字符串便是此选项值。如果主机集中设置了基于此选项的元数据 Key，则请求中的镜像也应该设置相应的元数据属性，这样 scheduler 才会将请求调度到对应主机集的主机上。此选项仅在使用 FilterScheduler 和其子类时才有效，如果使用其他的 scheduler，则此选项不会生效，此外，要使此选项生效，则必须启用过滤器 aggregate_image_properties_isolation
aggregate_image_properties_isolation_separator=.	字符串值，当启用 aggregate_image_properties_isolation 过滤器时，元数据 Key 由 aggregate_image_properties_isolation_namespace 配置选项值和一个分隔符号构成元数据 Key 命名空间，此选项的作用便是定义这个分隔符，其默认是圆点号（'.'）。此选项仅在使用 FilterScheduler 和其子类时才有效，如果使用其他的 scheduler，则此选项不会生效，此外，要使此选项生效，则必须启用过滤器 aggregate_image_properties_isolation
baremetal_scheduler_default_filters = RetryFilter, AvailabilityZoneFilter, ComputeFilter, ComputeCapabilitiesFilter, ImagePropertiesFilter, ExactRamFilter, ExactDiskFilter, ExactCoreFilter	列表值，此选项用于指定裸机过滤时使用的过滤器列表，选项值是字符串列表，每个字符串代表一个过滤器名称，在实际使用中，会按顺序执行列表中的过滤器，因此将条件最苛刻的过滤前放置在前可以使得过滤过程更高效。此选项仅在使用 FilterScheduler 和其子类时才有效，如果使用其他的 scheduler，则此选项不会生效，此外，要使此选项生效，则必须启用过滤器 scheduler_use_baremetal_filters
cpu_allocation_ratio = 0.0	浮点值，此选项代表虚拟 CPU 到物理 CPU 的分配比率，并将影响到全部 CPU 相关的过滤器。此选项值可以在每个计算节点上独立设置，但是如果在计算节点上将其设为 0.0，则将会使用部署 scheduler 服务节点上的值（通常是控制节点），并且会使用默认值 16.0
disk_allocation_ratio = 0.0	浮点值，此选项用于设置虚拟磁盘到物理磁盘的分配比率，disk_filter.py 脚本将会根据此参数值计算节点可用磁盘空间是否符合请求实例磁盘需求。如果此选项值大于 1，则意味着节点可用磁盘空间会被超额使用，可以将此选项值设置为 0.0 至 1.0 之间，以预留一定的节点磁盘空间用于存储其他信息。此选项可以在每个计算节点上单独设置，当设置为 0.0 时，部署 scheduler 服务节点上的此参数值会被使用，并且默认使用值为 1.0
disk_weight_multiplier = 1.0	浮点值，使用磁盘剩余空间计算权重时的权重因子，此选项值可以为正数也可以为负数，负值意味着实例会被收缩（stack）到某台主机上，而不是分散（spread）到多台主机上
io_ops_weight_multiplier = -1.0	浮点值，负载权重因子，用于根据节点负载计算节点的权重值。默认为负值，负值表示将实例创建在负载较轻的主机上，而正值表示创建在负载较重的主机上。此选项仅在使用 FilterScheduler 和其子类时才有效，并且此选项仅在使用 IoOpsWeigher 的情况下才生效
isolated_hosts =	列表值，当需要限制某些镜像只能运行在特定主机上时，便可使用此选项来设置这些主机，选项值为主机名组成的列表。此选项仅在使用 FilterScheduler 和其子类时才有效，并且此选项仅在使用 IsolatedHostsFilter 的情况下才生效
isolated_images =	列表值，当需要限制某些镜像只能运行在特定主机上时，便可使用此选项来设置这些镜像，选项值为镜像 UUID 名组成的列表。此选项仅在使用 FilterScheduler 和其子类时才有效，并且此选项仅在使用 IsolatedHostsFilter 过滤器的情况下才生效

(续)

<code>max_instances_per_host = 50</code>	<p>整数，如果需要设置主机允许运行的最大实例数目，便可通过此参数进行设置。如果主机上运行的实例数目已近达到此选项的设置值，则 <code>num_instances_filter</code> 过滤器将会拒绝该主机。选项有效值为正数，如果设置为 0，则 <code>num_instances_filter</code> 过滤器会拒绝全部主机。此选项仅在使用 <code>FilterScheduler</code> 和其子类时有效，并且此选项仅在使用 <code>num_instances_filter</code> 过滤器的情况下才生效</p>
<code>max_io_ops_per_host = 8</code>	<p>整数，此选项用于限定主机上可以有效执行 IO 操作（<code>build</code>, <code>resize</code>, <code>snapshot</code>, <code>migrate</code>, <code>rescue</code>, or <code>unshelve</code>）的实例数目，超过此选项值的主机将不能创建实例。此选项仅在使用 <code>FilterScheduler</code> 和其子类时有效，并且此选项仅在使用 <code>io_ops_filter</code> 过滤器的情况下才生效</p>
<code>ram_allocation_ratio = 0.0</code>	<p>浮点值，虚拟内存到物理内存的分配比率配置选项，此选项将会影响到全部基于内存的过滤器此选项值可以在每个计算节点上独立设置，但是如果在计算节点上将其设为 0.0，则将会使用部署 <code>scheduler</code> 服务节点上的值（通常是控制节点），并且会使用默认值 1.5</p>
<code>ram_weight_multiplier = 1.0</code>	<p>浮点值，内存权重因子，用于计算基于内存的主机权重。如果为正值，则 <code>scheduler</code> 将认为具有最大可用内存的节点最优，如果为负值，则是可用内存最小的节点最优。另外的理解方式是，正值意味着调度器倾向于将实例分布到更多的主机上，而负值则意味着尽量将实例创建到可用的主机上，直到资源不够使用为止。此选项仅在使用 <code>FilterScheduler</code> 和其子类时有效，并且此选项仅在使用 <code>RAMWeigher</code> 的情况下才生效</p>
<code>reserved_host_disk_mb = 0</code>	<p>整数，为主机预留的磁盘空间大小，单位为 MB</p>
<code>reserved_host_memory_mb = 512</code>	<p>整数，为主机预留的内存空间大小，单位为 MB</p>
<code>restrict_isolated_hosts_to_isolated_images = True</code>	<p>布尔值，此选项用于确定 <code>isolated_hosts filter</code> 过滤器是否允许未隔离镜像运行在隔离主机上，即是否允许没有进行元数据设置的普通镜像运行在特定设置了元数据的主机上。默认值为 <code>true</code>，表明未隔离镜像不被允许在隔离主机上运行，当设置为 <code>false</code> 时，未隔离镜像可以运行在隔离主机和未隔离主机上。此选项仅在使用 <code>FilterScheduler</code> 和其子类时有效，并且此选项仅在使用 <code>IsolatedHostsFilter</code> 过滤器的情况下才生效</p>
<code>scheduler_available_filters = ['nova.scheduler.filters.all_filters']</code>	<p>多值选项，该选项值是 <code>Nova scheduler</code> 可能使用到的过滤器类型无序列表，虽然只有在 <code>scheduler_default_filters</code> 选项中指定的过滤器才会被用到，但是任何出现在 <code>scheduler_default_filters</code> 选项中的过滤器必须事先被包含在此选项列表值中，默认情况下，此选项列表值包含了 <code>Nova</code> 自带的全部过滤器</p>
<code>scheduler_default_filters = RetryFilter, AvailabilityZoneFilter, RamFilter, DiskFilter, ComputeFilter, ComputeCapabilitiesFilter, ImagePropertiesFilter, Server-Group Anti Affinity Filter, Server Group Affinity Filter</code>	<p>列表值，此选项值是 <code>scheduler</code> 用于过滤主机的全部可用过滤器名称有序列表，选项值中的过滤器会按照先后顺序执行，因此确保条件苛刻的过滤器首先被使用可以提高整个过滤过程的效率。此外，任何出现在此选项中的过滤器都应该被包含在 <code>scheduler_available_filters</code> 中，否则异常 <code>SchedulerHostFilterNotFound</code> 会被抛出</p>
<code>scheduler_driver = filter_scheduler</code>	<p>字符串，<code>scheduler</code> 所使用的驱动类名称，如果此选项没有指定任何值，则 <code>filter_scheduler</code> 被使用，此选项目前仍然支持即将丢弃的 <code>python</code> 全路径形式值，如 <code>"nova.scheduler.filter_scheduler.FilterScheduler"</code>，但是在 <code>Newton</code> 版本后将不再支持，此选项允许的值除了 <code>filter_scheduler</code>，还包括 <code>caching_scheduler</code>、<code>chance_scheduler</code> 和 <code>fake_scheduler</code></p>

(续)

<code>scheduler_driver_task_period = 60</code>	整数值，此选项用于控制 scheduler 中的周期性任务多久执行一次。每个任务周期执行的具体任务由特定的 scheduler 来确定，如果此选项值大于 Nova 服务的 <code>service_down_time</code> 选项值，Nova 可能会发出 scheduler 已停止的信息，这是因为 scheduler 驱动负载发送心跳信号，并且发送频率由此选项值确定。由于不同 scheduler 的工作方式不一致，因此请确保针对特定的 scheduler 进行此选项值的测试
<code>scheduler_host_manager = host_manager</code>	字符串，表示 scheduler 使用的 host manager，默认值是 <code>host_manager</code> ，除此之外，在 Mitaka 版本发行后，还可以是另外一个值 <code>ironic_host_manager</code> 。此选项也支持全路径参数值形式，如 “ <code>nova.scheduler.host_manager.HostManager</code> ”，但是在 N 版本之后将不再支持
<code>scheduler_host_subset_size = 1</code>	整数值，此选项表示将会从 N 个最佳主机中随机选择一个主机进行请求实例的创建，此处的 N 即是该选项值。有效值为大于等于 1 的整数，任何小于 1 的值都会被看成 1。当此选项值大于 1 时，可以降低多个 scheduler 进程同时选中相同最佳主机的概率，通过从 N 个最佳主机中随机选取一个主机可以更好地满足请求，降低多个请求同时调度时产生突出的机率。但是此选项值设置越高，所带来的优势就会变得越小
<code>scheduler_instance_sync_interval = 120</code>	整数值，等待向 scheduler 发送实例 UUIDs 列表的实际间隔，向 scheduler 发送 UUIDs 的目的在于验证 Nova 与其实例是同步的。如果选项 <code>optionscheduler_tracks_instance_changes</code> 值为 <code>False</code> ，则此选项没有意义
<code>scheduler_json_config_location =</code>	字符串，scheduler 的 json 配置文件绝对路径
<code>scheduler_manager = nova.scheduler.manager.SchedulerManager</code>	字符串，scheduler 管理器的全路径名称，后续版本即将丢弃此选项
<code>scheduler_max_attempts = 3</code>	整数值，在 scheduler 认为实例创建调度失败并非由正常的竞争冲突而是由其他问题导致之前，重复尝试调度实例的次数。当达到此选项值时，会抛出 <code>MaxRetriesExceeded</code> 异常，并且实例会被置为 <code>error</code> 状态，有效值为大于等于 1 的整数值
<code>scheduler_topic = scheduler</code>	字符串，此选项表示 scheduler 服务监听的消息队列 topic 名称。当 scheduler 服务启动并配置消息队列和有 RPC 调度 scheduler 服务时候都会用到此选项值，正常情况下几乎不会更改此选项值
<code>scheduler_tracks_instance_changes = True</code>	布尔值，为了对主机进行过滤和加权，调度器 scheduler 可能会需要用到主机上的实例信息，最常见的需要主机实例信息情况就是在启用 <code>affinity filters</code> 或 <code>anti-affinity filters</code> 过滤器时，如果没有过滤器需要用到主机实例信息，可以将此选项设为 <code>false</code> 以提升性能
<code>scheduler_use_baremetal_filters = False</code>	布尔值，设置此选项为 <code>true</code> 将会通知 nova scheduler 使用在选项 <code>baremetal_scheduler_default_filters</code> 中指定的过滤器，如果不需要进行裸机节点调度，则保持此选项值为 <code>false</code> 即可
<code>scheduler_weight_classes = nova.scheduler.weights.all_weighters</code>	列表值， <code>weigher</code> 类名列表。此选项默认值包含了所有 Nova 自带的 <code>weighters</code>
<code>soft_affinity_weight_multiplier = 1.0</code>	浮点值，在启用 <code>ServerGroupSoftAffinityWeigher</code> 时，用于计算主机权重值的权重因子，对于此选项值，仅有正数值是有意义的
<code>soft_anti_affinity_weight_multiplier = 1.0</code>	浮点值，在启用 <code>ServerGroupSoftAntiAffinityWeigher</code> 时，用于计算主机权重值的权重因子，对于此选项值，仅有正数值是有意义的

8.6 Nova 实例创建

8.6.1 Nova 实例创建流程

在 OpenStack 各个服务组件中，计算服务 Nova 是最核心也是最复杂的服务项目之一，其复杂性在虚拟机实例创建时便可见一斑。在 OpenStack 中，创建实例主要有两种方式：Nova 客户端命令行和 Dashboard 用户接口 GUI 界面。无论是哪种方式，其实例创建过程的后端调用流程本质上是相同的，都要经过 Nova 内部组件之间和 Nova 与其他服务项目之间的频繁交互才能完成实例的创建工作。正常情况下，要完成一个实例的创建，需要参与的 OpenStack 服务项目除了 Nova 之外，还有 Dashboard、Keystone、Glance、Cinder 和 Neutron 等项目，其中，Nova、Keystone、Glance 和 Neutron 为必须项目，而 Dashboard 和 Cinder 为非必须。此外，在 Nova 内部，参与实例创建的组件包括 Nova-api、Nova-conductor、Nova-scheduler、Nova-compute、Nova-consoleauth、Nova-novncproxy、Nova-cert 和 Nova-objectstore 等，其中，Nova-api、Nova-conductor、Nova-scheduler、Nova-compute 为必须使用到的组件。如果要使用虚拟网络控制台 VNC 与实例交互，则 Nova-consoleauth 和 Nova-novncproxy 是必须的；如果使用 AWS 风格 API，则 Nova-cert 和 Nova-objectstore 是必须的。

在 OpenStack 中创建实例的大致流程为：用户通过 Dashboard 界面或命令行发起实例创建请求，Keystone 从请求中获取用户相关信息并进行身份验证；验证通过后，用户获得认证 Token，实例创建请求进入 Nova-api；在向 Keystone 验证用户 Token 有效后，Nova-api 将请求转入 Nova-scheduler；Nova-scheduler 进行实例创建目的主机的调度选择，目标主机选取完成后，请求转入 Nova-compute；Nova-compute 与 Nova-conductor 交互以获取创建实例的信息，在成功获取实例信息后，Nova-compute 分别与 Glance、Neutron 和 Cinder 交互以获取镜像资源、网络资源和云存储资源；一切资源准备就绪后，Nova-compute 便通过 Libvirt API 与具体的 Hypervisor 交互并创建虚拟机。OpenStack 实例创建的完整流程如图 8-39 所示，下面将对 OpenStack 实例创建中服务项目之间的交互和项目组件内部的工作机制进行详细介绍。

1. Keystone 工作流程

任何终端用户在与 OpenStack 的服务项目交互时，第一步便是进行身份认证与授权，而 Keystone 负责 OpenStack 全部项目的认证与授权任务。当 Nova 客户端发起实例创建请求时，Nova 客户端便与 Keystone 进行交互以获取授权 Token，如果 Nova 客户端通过 Keystone 的身份验证，则 Keystone 为其生成授权 Token，并将此 Token 存储到后端数据库中。Keystone 的后端数据库可以是 SQL 数据库，也可以是如 Memcache 这样的缓存系统。为了快速响应后续 OpenStack 服务组件发起的 Token 核实过程，建议尽量使用基于内存的缓存系统作为 Keystone 的后端存储数据库。Keystone 的身份验证过程可以在本地进行，也

可以通过 LDAP 服务器进行身份验证，图 8-40 为 Nova 客户端与 Keystone 交互进行身份验证和 Token 颁发过程（这里 Keystone 使用 LDAP 进行身份验证）。

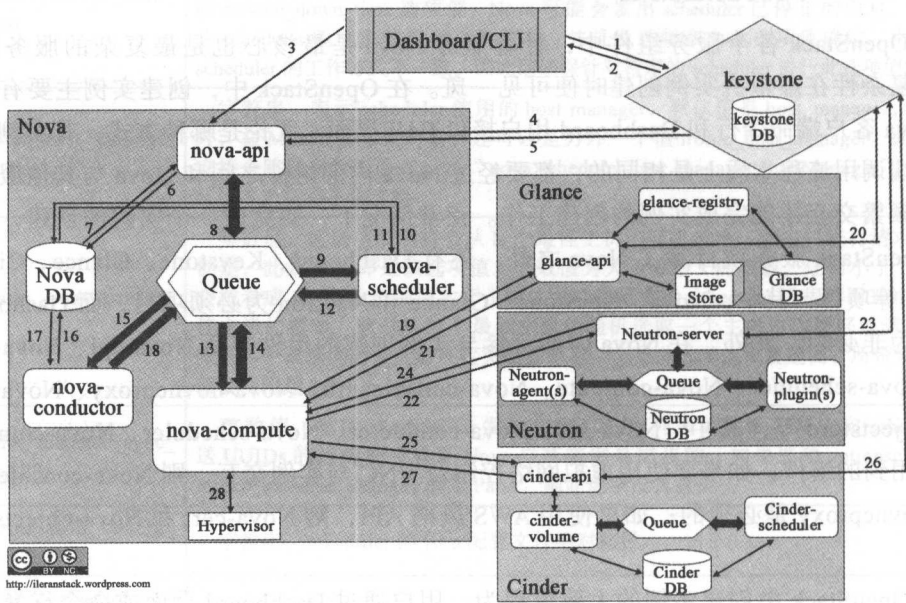


图 8-39 OpenStack 实例创建过程

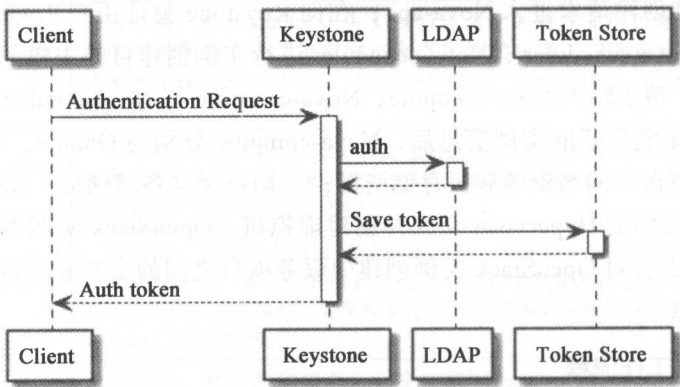


图 8-40 Nova 客户端身份验证与授权

2. Nova-api 工作流程

Nova 客户端在通过 Keystone 的验证并获取 Token 后，便向 Nova-api 发出实例创建的请求，Nova-api 接收到请求后向 Keystone 验证 Token 的有效性，验证成功后，Nova-api 开始判断实例创建请求所携带的参数是否有效合法，如检查虚拟机名字是否符合命名规范，使用的虚拟机模板 flavor_id 在数据库中是否存在，使用的镜像 image_uuid 是否是有效的 uuid 格式等。检查 instance、vCPU、RAM 的数量是否超出了配置文件中的限制，

通常每个 Project 拥有的资源都是有限的，如创建虚拟机的个数、vCPU 个数、内存大小和 volume 个数等，这些限制是管理员通过与 Project 相关的 Quota 来设置的，如 `quota_instances`、`quota_cores`、`quota_ram`、`quota_volumes` 等。如果管理员未做设置更改，则默认情况下所有 Project 拥有相等的资源数量。此外，Nova-api 还会检查相关 metadata 的长度是否超过限制，`inject_files` 的数目是否超过限制，一般情况下这些参数都为空，因此都能通过检查。实例请求中的网络、镜像和 flavor 也是 Nova-api 主要的检查对象，其主要检查请求中的 `network` 是否存在且可用以及 `image` 和 `flavor` 是否存在且可用，同时还会检测请求中 `flavor` 的磁盘是否满足 `image` 需求等。当所有资源检查都通过后，Nova-api 便在 nova 数据库中初始化虚拟机相关的记录 (initial entry)，主要包括 `instance`、`block_device_mapping` 和 `quota` 等记录，并将 `instance` 记录中的 `vm_states` 字段设为 `building`，`task_state` 字段设为 `scheduling`。之后，Nova-api 调用 Nova-conductor 并将请求传递到消息队列 (MQ) 中。由于 Nova-api 将请求以 RPC cast 的方式发送给 Nova-conductor，而 `cast()` 方法发送的请求并不会返回消息，因此 Nova 客户端此时只会接收到请求已被接受的返回，但是具体的虚拟创建过程还在后台继续执行，而 Nova 客户端可以查到的虚拟机 `vmstate` 为 `building`，`task_state` 为 `scheduling`。Nova-api 在实例创建过程中的工作流程如图 8-41 所示。

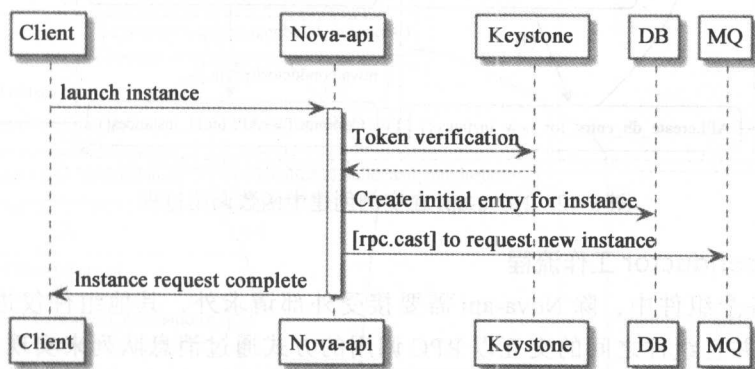


图 8-41 Nova-api 在实例创建中的工作流程

图 8-41 中 Nova-api 工作流程对应的函数调用和执行顺序如图 8-42 所示。客户端请求经过身份验证后，`nova/api/openstack/compute/servers.py` 中的 `ServersController.create()` 方法将被调用，进而调用 `nova/compute/api.py` 中的 `API.create()` 方法（此 API 类中除了 `create()` 方法，还定义了各种请求参数检查方法，如 `_check_requested_networks()` 方法、`_check_requested_image()` 方法和 `_check_requested_secgroups()` 方法等，同时还定义了与虚拟机相关的全部操作方法如 `reboot()` 方法、`_soft_reboot()` 方法、`_hard_reboot()` 方法和 `rebuild()` 方法等）。`API.create()` 方法随后调用 `_create_instance()` 方法，此方法主要负责参数处理，随后调用 `_provision_instance()` 方法，并在数据库中初始化虚拟机相关的记录。然后，通过 `nova/conductor/api.py` 中的 `ComputeTaskAPI.build_instances()` 将请求下发给 Nova-

conductor。而 build_instances 又调用 nova/conductor/rpcapi.py 中的 build_instances(), 以将请求传递到 RabbitMQ 服务器的消息队列中, 至此, Nova-api 的工作执行完成, 并告知 Nova 客户端请求已被接受, 客户端可以进行其他操作, 实例创建过程在后台继续进行。

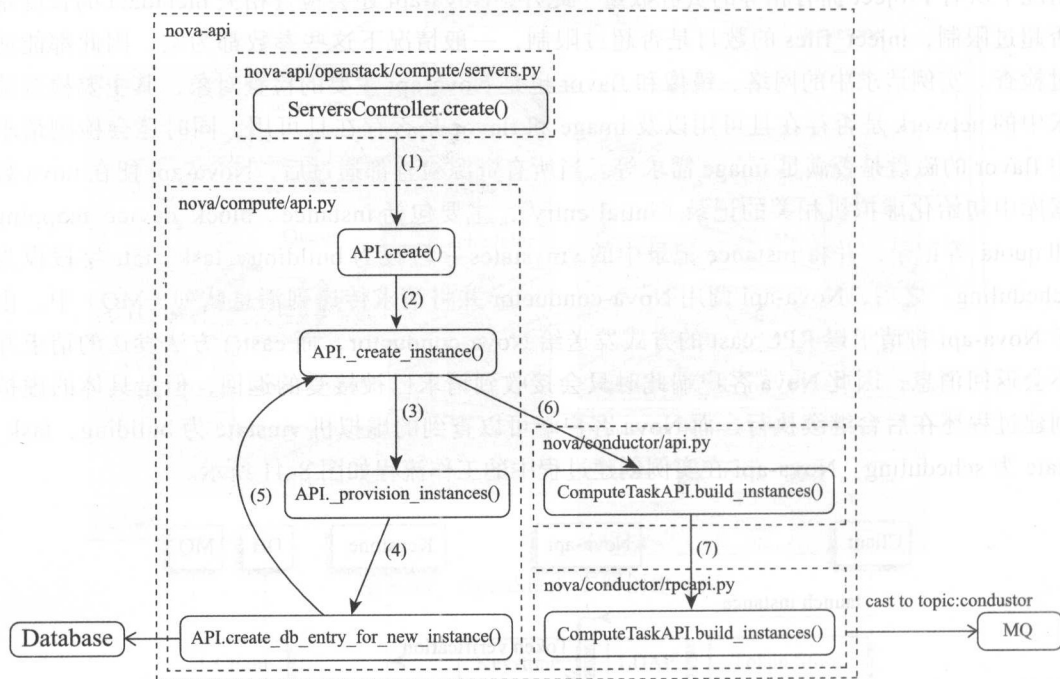


图 8-42 Nova-api 在实例创建中函数调用过程

3. Nova-conductor 工作流程

在 Nova 各个组件中, 除 Nova-api 需要接受外部请求外, 其他组件仅进行彼此之间的交互调用, 并且组件之间的交互以 RPC 调用的方式通过消息队列来实现。由于 Nova-conductor 提供了 build_instances() 这个 RPC 方法, 因此一直处于消息队列监听状态, 一旦监听的队列有消息进入, Nova-conductor 便开始执行 build_instances() 方法。Nova-conductor 还向 Nova-scheduler 发出 RPC call 调用, 并要求其返回计算节点调度结果, 在收到 Nova-scheduler 的调度结果后, Nova-conductor 的 build_instances() 方法将请求传递到 Nova-compute 的消息队列中, 如图 8-43 所示。

图 8-44 是 Nova-conductor 组件的内部函数执行流程, Nova-conductor 从 topic:conductor 消息队列中提取 Nova-api 传递进来的消息, 并执行 build_instances() 方法, 此方法经过一系列函数调用, 向 topic:scheduler 队列发出 RPC call 调用消息, Nova-scheduler 在获取 topic:scheduler 对象消息后, 经过内部 Filter 和 Weigh 过程, 调度出一个或多个最优计算节点, 并将经过返回给消息队列, Nova-conductor 通过消息队列获取 Nova-scheduler 返回的计算节点列表值, 并通过 build_and_run_instance() 方法将实例创建请求发送到 Nova-compute

点的调度选取，同时将调度结果传递给消息队列。为了获取创建实例的目标主机，Nova-conductor 会向 Nova-scheduler 发起 RPC call 调用，并调用 Nova-scheduler 的 `select_destinations()` 方法；Nova-scheduler 在消息队列中接收到调用消息后，根据 `nova.conf` 配置文件中关于 `Filters` 和 `Weight` 的配置参数对计算节点主机列表进行过滤和加权调度，在这个过程中，Nova-scheduler 需要访问数据库以获取节点相关的信息；在获取信息后，Nova-scheduler 便开始进行节点调度，默认使用的调度驱动为 `FilterScheduler`，调度完成之后，将节点调度结果传递到消息队列，以完成 Nova-conductor 的 RPC call 调用过程。Nova-scheduler 的工作流程如图 8-45 所示。

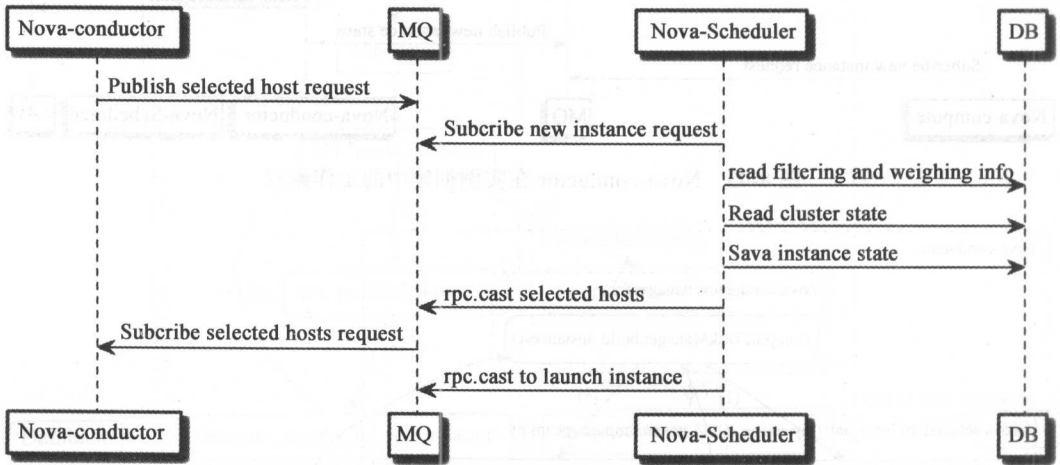


图 8-45 Nova-scheduler 在实例创建中的工作流程

图 8-46 为 Nova-scheduler 在实例创建过程中的内部函数调用流程。Nova-scheduler 从 `topic:scheduler` 消息队列获取调用信息后，开始执行 `select_destinations()` 方法，由于 Nova 通常采用默认的 `FilterScheduler` 调度器，因此 `FilterScheduler` 的 `select_destinations()` 和 `_scheduler()` 方法被执行，`_scheduler()` 函数最终会调用获取主机信息的函数和各种节点过滤器以及加权方法以对计算节点进行过滤和计算权重值，并将最终调度选取的计算节点信息传递到消息队列，返回给 Nova-conductor。

5. Nova-compute 工作流程

Nova-compute 是 Nova 服务项目中最复杂和最关键的组件，运行 Nova-compute 的节点称为计算节点，通常 Nova-compute 部署在独立的计算节点上，并与 Nova 项目的其他组件分开部署。在实例创建过程中，Nova-compute 随时监听 `topic:compute-computeN` (`computeN` 为计算节点名称) 消息队列，在监听到 Nova-conductor 传递的实例创建请求后，Nova-compute 开始启动内部工作流程以响应实例创建请求。在 Nova-api 的工作流程中，请求中创建实例所需的镜像、网络 and 存储等资源已经做过有效性和可用性的检查，因此 Nova-compute 将直接与 Glance 交互以获取镜像资源，与 Cinder 交互获取块存储资源。

如果配置了 Ceph 块存储服务或对象存储服务，Nova-compute 还会与 Ceph RADOS 进行交互以访问 Ceph 存储集群。Nova-compute 与 Glance、Cinder 和 Ceph 的交互访问流程如图 8-47 所示。

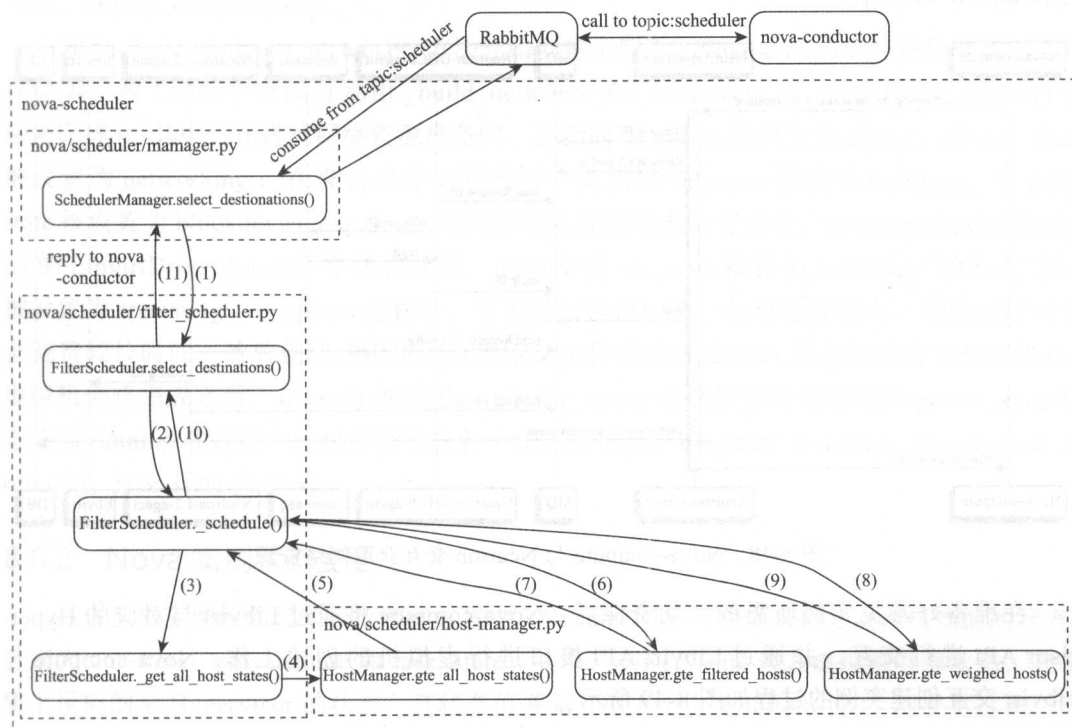


图 8-46 Nova-scheduler 在实例创建中的函数调用过程

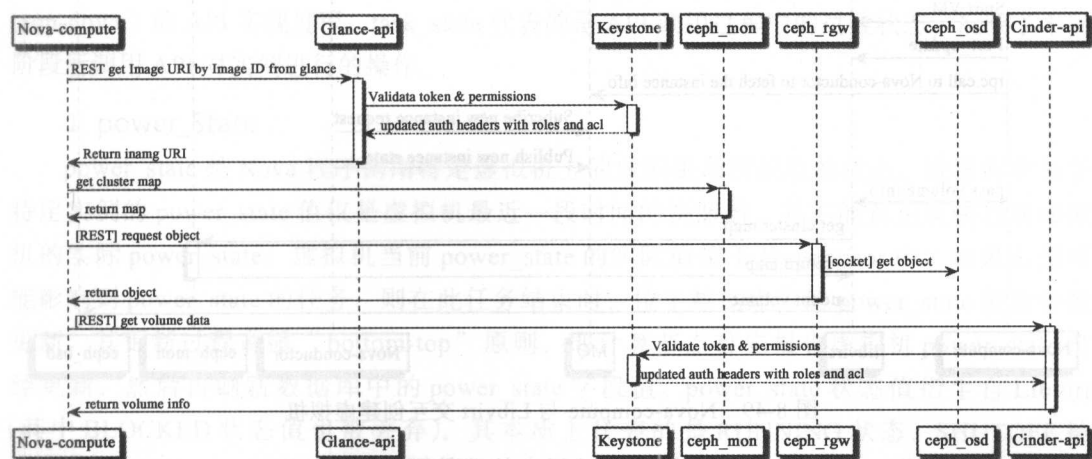


图 8-47 Nova-compute 与 Glance、Cinder 和 Ceph 的交互访问流程

在获取镜像和存储资源后，Nova-compute 还需与 OpenStack 的网络服务项目 Neutron 进行交互访问以获取网络资源。由于网络资源的有效性和可用性已经在 Nova-api 工作流程中完成，这里主要是获取虚拟机的 Fixed IP 等网络资源。Nova-compute 与 Neutron 的交互过程如图 8-48 所示。

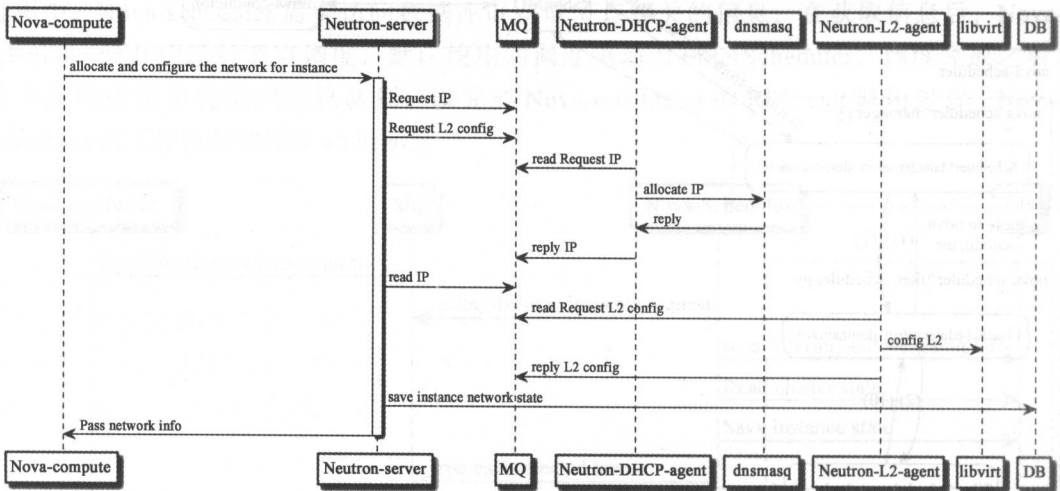


图 8-48 Nova-compute 与 Neutron 交互获取网络资源

在准备好常见实例所需的一切资源后，Nova-compute 将通过 Libvirt 与对应的 Hypervisor API 进行交互，并通过 Libvirt API 接口进行虚拟机的创建工作。Nova-compute 与 Libvirt 交互创建实例的过程如图 8-49 所示。

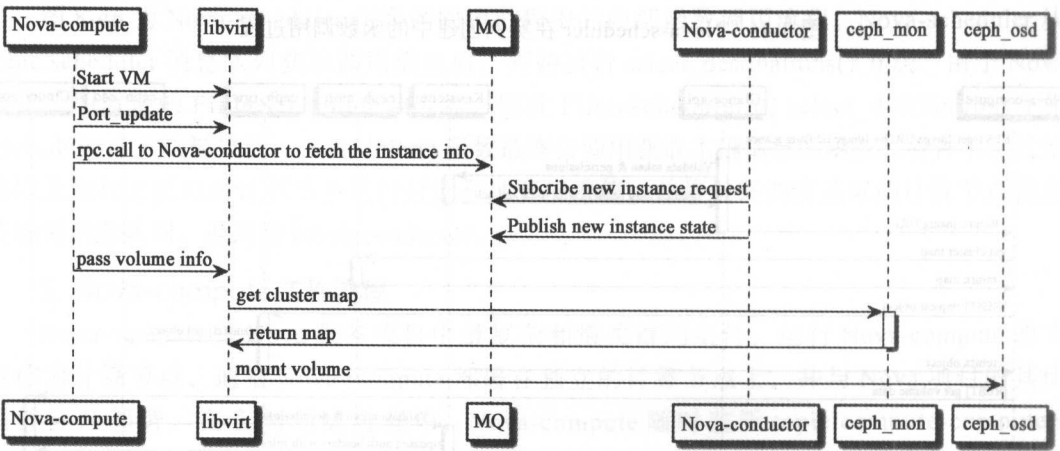


图 8-49 Nova-compute 与 Libvirt 交互创建虚拟机

图 8-50 为 Nova-compute 在实例创建过程中的函数调用过程。Nova-compute 接收到消息队列中由 Nova-conductor 发起的 RPC cast 调用请求，该请求调用 nova/compute/manager.

py 中的 `ComputeManager.build_and_run_instance()` 方法，该方法继而调用 `ComputeManager` 类中的 `_build_and_run_instance()` 方法以尝试创建实例；这里创建实例不一定成功，如果创建失败并且是因为 `Scheduler` 调度结果的原因，则会发起 `Re-scheduler` 操作以重新调度计算节点，并再次尝试创建实例。

进入 `_build_and_run_instance()` 方法开始创建实例后，`Nova-compute` 便开始准备网络和存储等资源（这两个资源分别由 `_build_networks_for_instance()` 和 `_prep_block_device()` 方法来完成）。其中，在进行网络资源准备时，实例的 `vm_state` 被设为 `building`，而 `task_state` 被设置为 `networking`；在准备块存储资源时，实例的 `vm_state` 被设为 `building`，而 `task_state` 被设置为 `blockdevicemapping`。在实例所需资源均准备完成后，`nova/virt/libvirt/driver.py` 的 `LibvirtDriver.spawn()` 方法被调用，此时实例 `vm_state` 被设为 `building`，而 `task_state` 被设置为 `spawning`。在 `spawn` 过程中，首先进行镜像加载，如果镜像较大，则此过程可能会花费较长时间，然后创建虚拟机的资源定义文件并通过 `Libvirt` 发起创建虚拟机的操作。虚拟机创建完成之后，`spawn()` 将调用 `_wait_for_boot()` 方法以等待虚拟机的 `power_state` 状态变为 `running` 才返回，此时虚拟机的各个状态应该是，`vm_state` 为 `active`，`power_state` 为 `running`，`task_state` 为 `none`。

8.6.2 Nova 实例状态变更

在虚拟机实例的创建和运行维护过程中，`Nova` 使用三个变量来描述虚拟机当前的状态，分别为 `vm_state`、`power_state` 和 `task_state`。其中，`power_state` 代表虚拟机电源状态，其本质上反应的是 `Hypervisor` 的状态，其状态值遵循“由下至上（`bottom-up`）”的变更过程，即先是底层计算节点上 `Hypervisor` 状态变更，然后上层数据库对应值变更；`vm_state` 反应的是基于 `API` 调用的一种稳定状态，其状态变更比较符合用户的逻辑思维，是一种由上至下（`top-down`）的 `API` 实现过程；`task_state` 代表的是 `API` 调用过程中的过渡状态，反映了不同阶段所调用 `API` 对实例进行的操作。

1. power_State

`power_state` 是 `Nova` 程序调用特定虚拟机上的虚拟驱动所获取的状态。数据库中关于特定实例的 `power_state` 值仅是虚拟机最近一段时间的快照值，并不能真正反映当前虚拟机的实际 `power_state`，虚拟机当前 `power_state` 的实际值位于 `Hypervisor` 中。如果出现可能影响到 `power_state` 的任务，则在此任务结束时，位于数据库中的 `power_state` 值就会被更新。其更新过程遵循“`bottom-top`”原则，即计算节点首先报告虚拟机 `power_state` 已经更新，然后再刷新数据库中的 `power_state` 字段值。`power_state` 状态值衍生自 `Libvirt`（其中 `BLOCKED` 状态值也被丢弃），其本质上代表的是 `RUNNING` 状态，`SHUTOFF` 被映射为 `SHUTDOWN` 状态，`FAILED` 被映射为 `NOSTATE` 状态，因此 `power_state` 目前有 `RUNNING`、`SHUTDOWN` 和 `NOSTATE` 三种状态。

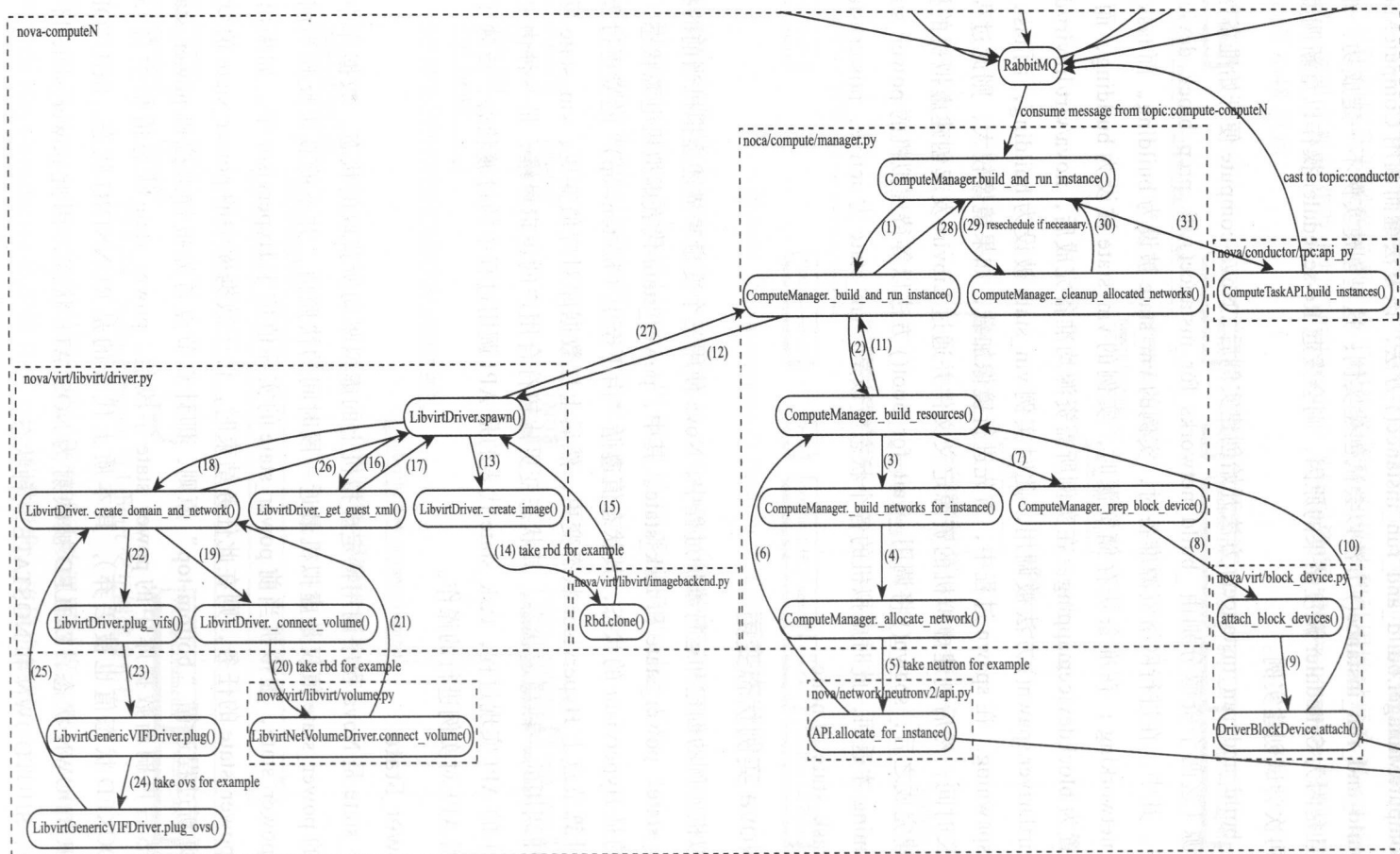


图 8-50 Nova-compute 在实例创建中的函数调用过程

2. vm_State

vm_state 状态值是对虚拟机当前稳定状态的描述，而不是过渡状态。也就是说，如果针对特定虚拟机已经没有 API 继续调用，则应该用 vm_state 来描述此时虚拟机的稳定状态。例如，当 vm_state 状态值为 ACTIVE，则表示虚拟机正常运行。再如 SUSPENDING 状态，其表示虚拟机正处于挂起过程中，并且在几秒之后会过渡到 SUSPENDED 状态，因此这是一种过渡状态，即 SUSPENDING 应该属于 task_state 的状态值。vm_state 状态值更新的前提是针对此虚拟机有任务发生，并且任务成功完成，同时 task_state 被置为 NOSTATE。如果没有 API 调用发生，则 vm_state 状态值绝对不会改变；如果对虚拟机的操作任务失败，但是成功回滚（rollback），则 vm_state 状态值也不会改变；如果不能回滚成功，则 vm_state 被置为 ERROR 状态。

vm_state 状态与 power_state 状态之间没有必然的对应关系，即不能由 vm_state 的某一状态值推导出 power_state 的状态，反之亦然。当有针对虚拟机的操作任务正在执行时，power_state 与 vm_state 的状态值很可能不协调：出现不协调的主要原因是 vm_state 仅代表虚拟机的稳定状态，而在任务执行中，虚拟机的状态一直在过渡，而且不能及时同步更新。如果没有任务正在执行，则 vm_state 与 power_state 不应该冲突，除非出现错误或任务失败，此时就需要具体问题具体分析以解决二者之间的不一致。常见的不一致有以下几种：

- ❑ power_state 为 SHUTDOWN，而 vm_state 为 ACTIVE：这种情况最可能的原因是虚拟机内部执行 shutdown 命令时出现异常，解决这个问题一个简单粗暴的方法就是手动调用 stop() API，之后，vm_state 应该变为 STOPPED。
- ❑ power_state 为 RUNNING，而 vm_state 为 HARD_DELETE：如果出现这种情况，则表明用户已经发出删除虚拟机的命令，但是执行过程出错，可以尝试再次删除虚拟机。
- ❑ power_state 为 RUNNING，而 vm_state 为 PAUSED：出现这个情况，表明虚拟机在执行 pause() 之前出现了意外情况，这个问题的解决办法就比较多样，可以尝试将其设为 ERROR。

vm_state 有多个状态值，不同状态值表示虚拟机当前处于不同的稳定状态。有些状态值彼此之间可以互相转换，如 PAUSED 与 ACTIVE 状态；而有些状态值之间没有任何直接联系，如 PAUSED 与 STOPPED；没有联系的状态值之间只有借助第三个状态值才能转换，如 ACTIVE 状态值。图 8-51 为 vm_state 不同状态值之间的相互转换关系图，其中任何虚拟机状态都可以直接转移到 DELETE 和 ERROR 状态。

3. task_state

task_state 代表的是过渡状态，并且与调用的 API 函数密切相关，其状态值描述了虚拟机当前正在执行的任务。只有对虚拟机发起了操作任务时，才会有 task_state 状态值出现，而当 vm_state 处于稳定值时，task_state 通常为 NOSTATE。简单来说，当虚拟机在执行任

务时, `task_state` 便会有反映操作任务的状态值, 而如果没有执行任何任务, 则 `task_state` 一定是 `NOSTATE`。

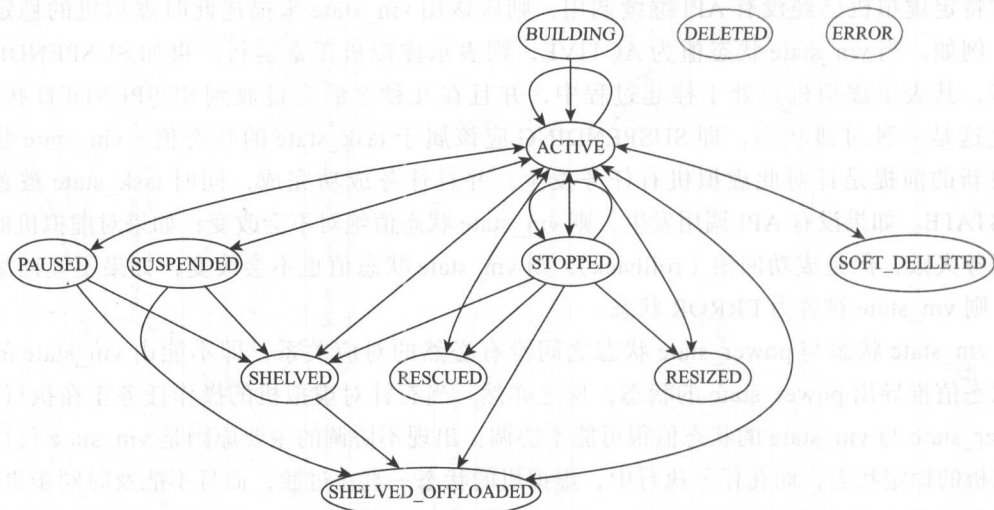


图 8-51 虚拟机状态转换

`task_state` 的一个特殊任务状态是 `FORCE_DELETE` (或 `HARD_DELETE`)。正常情况下, 任何时候用户都可以成功删除虚拟机, 删除虚拟机后便可使用更多受 Quota 限制的資源, 同时删除的虚拟机不会被计费, 但是虚拟机删除任务可能会因为某些原因而失败, 例如之前的任务堵塞或卡死而导致虚拟机删除失败, 删除虚拟机时虚拟驱动卡死或计算节点因为网络 / 硬件不可用导致操作失败等。失败的虚拟机删除任务意味着 `task_state` 状态值不能成功过渡到 `NOSTATE`, 而虚拟机状态值 `vm_state` 更新的前提是 `task_state` 从具体的任务状态过渡到 `NOSTATE`, 因此, 对于 `force_delete()` 任务, 不应该等到任务执行到计算节点并完成全部相关工作才更新 `vm_state` 的状态值为 `HARD_DELETE`, 而是应该发起 `force_delete()` 任务之后便立即更新 `vm_state` 状态值。也就是说, `force_delete()` 是个纯粹的数据库操作任务, 其对数据库中 `vm_state` 字段值刷新之后便结束, 而相应的虚拟机清除工作随后才进行。

当任务被确认为虚拟机上唯一执行的任务时, `task_state` 就会被设置为具体的状态值。虚拟机中每个正在执行的任务都会被分配一个与虚拟机相关的唯一 `task_id` (UUID 格式)。如果虚拟机已经有个 `task_id`, 则表明有任务正在执行, 要在任务执行中途更新 `task_state`, 就必须确保虚机的 `task_id` 匹配当前的 `task_id`, 否则当前执行的任务会被抢占 (目前只有 `force_delete` 任务能抢占)。通常在某个具体的任务执行期间, `task_state` 的值绝对不能改变。当一个任务执行完成后, `task_state` 被置为 `NOSTATE`, 而 `task_id` 被置为 `NONE`。`task_state` 的状态值名称通常是代表某个 API 方法动词的正在进行时 (“ing” 形式), 如表 8-10 所示, 因此, 从 `task_state` 的状态值便可推出虚拟机当前正在执行什么任务。

表 8-10 task_state 状态值及其任务

状 态 值	任 务	状 态 值	任 务
NONE	当前没有任何任务	REBUILDING	正在重新创建虚拟机
BUILDING	正在 building，实例创建起始阶段	POWERING_ON	正在 poweron 虚拟机
IMAGE_SNAPSHOT-TING	正在执行镜像快照	POWERING_OFF	正在 poweroff 虚拟机
IMAGE_BACKINGUP	正在进行镜像备份	RESIZING	正在 resize 虚拟机
UPDATING_PASS-WORD	正在更新 password	RESIZE_REVERTING	正在 resize 复原
PAUSING	正在暂停虚拟机	RESIZE_CONFIRMING	正在 resize 结果确认中
UNPAUSING	正在激活暂停中的虚拟机	SCHEDULING	正在进行节点调度
SUSPENDING	正在挂起虚拟机	BLOCK_DEVICE_MAPPING	正在进行块设备映射
RESUMING	正在激活挂起中的虚拟机	NETWORKING	正在进行网络设置
DELETING	正在删除虚拟机	SPAWNING	实例创建的最后阶段，正在 spawn
STOPPING	正在停止虚拟机	RESIZE_PREP	resize 准备
STARTING	正在启动虚拟机	RESIZE_MIGRATING	resize 迁移中
RESCUING	正在拯救虚拟机	RESIZE_MIGRATED	resize 迁移完成
UNRESCUING	正在撤销虚拟机拯救操作	RESIZE_FINISH	resize 任务完成
REBOOTING	正在重启虚拟机		

对任一实例，不管是在实例创建过程中，还是创建完成后的维护操作中，vm_state、power_state 和 task_state 总是同时存在，并共同决定了虚拟机的当前运行状态。图 8-52 为虚拟机创建过程中 vm_state、power_state 和 task_state 的状态值变更过程，可以看到，实例创建过程中，vm_state 状态由 Building 变为 Active，power_state 则由 NoState 变为 Running，经历状态值变更最多的是 task_state，其经历了 Scheduling、None、Networking、Block_Device_Mapping 和 Spawning 等状态值的变更。

不同的虚拟机状态 (vm_state)，意味着用户可以对虚拟机发起不同的操作。因此，用户在针对虚拟机进行各种操作时，一定要核实清楚当前虚拟机的 vm_state 状态值是什么 (因为虚拟机在特定的状态值下，仅允许执行特定的命令)。通常虚拟机状态与可执行的命令为一对一或一对多的关系，如当虚拟机状态为 Paused 时，只有 unpause 命令可以执行，当虚拟机状态为 Active 时，可以有 suspend、pause 和 rescue 等多个命令可以执行。表 8-11 列出了不同 vm_state 状态值下，用户可以对虚拟机执行的各种命令。

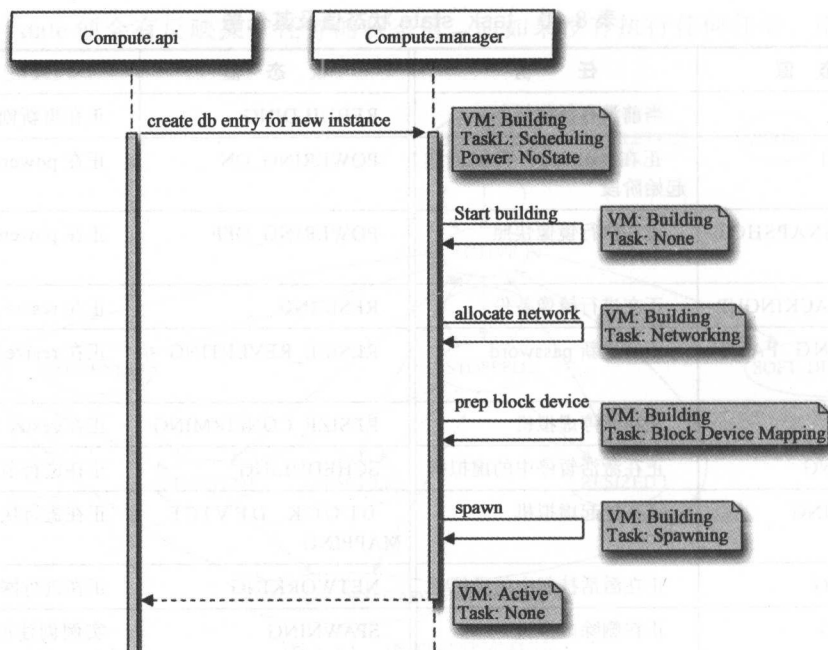


图 8-52 实例创建过程中的 vm_state、power_state 和 task_state 状态值

表 8-11 虚拟机状态与可执行命令对照表

虚拟机状态	可执行命令
Paused	unpause
Suspended	resume
Active	set admin password, suspend, pause, rescue, rebuild, soft delete, delete, backup, snapshot, stop, reboot, resize, revert resize, confirm resize
Shutoff	suspend, pause, rescue, rebuild, soft delete, delete, backup, start, snapshot, stop, reboot, resize, revert resize, confirm resize
Rescued	unrescue, pause
Stopped	rescue, delete, start
Soft Deleted	force delete, restore
Error	soft delete, delete
Building	delete
Rescued	delete, stop, reboot

虚拟机之间的状态可以相互转换，图 8-52 描述了这种状态转换关系。让虚拟机由一个或多个状态转换到另外一种或几种状态，需要对虚拟机发出相关的任务操作，或者说要让虚拟机最终处于某种状态，则需要对当前状态的虚拟机执行相应的操作命令。虚拟机当前状态与目标状态之间的对应关系可以是一对一或多对一，如当前为 Paused 状态的虚拟机通

过 `unpause` 命令可以变为 Active 状态，当前是 Active、Shutoff 和 Rescued 状态的虚拟机都可以通过 `pause` 命令变为 Paused 状态，而对于某些如 `backup` 或 `snapshot` 等备份命令则不会改变虚拟机状态。表 8-12 为虚拟机当前状态、目标状态和执行命令之间的对应关系。

表 8-12 虚拟机当前状态、目标状态和执行命令对照表

执行命令	虚拟机当前状态	任务状态	虚拟机目标状态
<code>pause</code>	Active, Shutoff, Rescued	Resize Verify, unset	Paused
<code>unpause</code>	Paused	N/A	Active
<code>suspend</code>	Active, Shutoff	N/A	Suspended
<code>resume</code>	Suspended	N/A	Active
<code>rescue</code>	Active, Shutoff	Resize Verify, unset	Rescued
<code>unrescue</code>	Rescued	N/A	Active
<code>set admin password</code>	Active	N/A	Active
<code>rebuild</code>	Active, Shutoff	Resize Verify, unset	Active
<code>force delete</code>	Soft Deleted	N/A	Deleted
<code>restore</code>	Soft Deleted	N/A	Active
<code>soft delete</code>	Active, Shutoff, Error	N/A	Soft Deleted
<code>delete</code>	Active, Shutoff, Building, Rescued, Error	N/A	Deleted
<code>backup</code>	Active, Shutoff	N/A	Active, Shutoff
<code>snapshot</code>	Active, Shutoff	N/A	Active, Shutoff
<code>start</code>	Shutoff, Stopped	N/A	Active
<code>stop</code>	Active, Shutoff, Rescued	Resize Verify, unset	Stopped
<code>reboot</code>	Active, Shutoff, Rescued	Resize Verify, unset	Active
<code>resize</code>	Active, Shutoff	Resize Verify, unset	Resized
<code>revert resize</code>	Active, Shutoff	Resize Verify, unset	Active
<code>confirm resize</code>	Active, Shutoff	Resize Verify, unset	Active

8.6.3 Nova 实例创建方法

实例创建是 OpenStack 中最为关键的环节，对于 OpenStack 而言，没有实例就等同于一堆没有任何价值的 IaaS 基础架构。实例创建可以认为是检验 OpenStack 关键功能是否正确可用的步骤，包括像 Cinder、Keystone、Nova、Glance 和 Neutron 等核心项目的部署配置都是为了可以提供对外服务的实例，因此，也可以认为实例创建是 OpenStack “最后”的 IaaS 操作。常见的实例创建有两种方式，分别为基于本地镜像的实例创建和基于可引导卷的实例创建，对应于传统 IT 架构就是本地磁盘操作系统和 SAN BOOT 形式的操作系统。在实例创建前，用户需要收集创建实例所需的资源，并确保这些资源都是有效可用的，如

镜像、密钥、网络和虚机模板等。实例创建所需资源的收集分为以下几个方面：

(1) 收集虚拟机模板信息

模板 (Flavor) 定义了实例所需的 CPU、内存、磁盘和临时存储等信息, Nova 默认自带 5 个 Flavor, 此外, 用户可以根据需求创建自己的 Flavor。创建实例前, 用户需要查看 Nova 中已有的 Flavor, 并记住其 ID 以便在实例创建中使用。模板查看命令如下:

```
[root@controller1 ~]# nova flavor-list
```

ID	Name	Memory_MB	Disk	Ephemeral	Swap	VCPUs	RXTX_Factor	Is_Public
1	m1.tiny	512	1	0		1	1.0	True
2	m1.small	2048	20	0		1	1.0	True
3	m1.medium	4096	40	0		2	1.0	True
4	m1.large	8192	80	0		4	1.0	True
5	m1.xlarge	16384	160	0		8	1.0	True

(2) 收集镜像信息

Nova 或 Glance 并无默认镜像, 需要用户自己制作符合自己需求的镜像并将其上传到 Glance 中, OpenStack 官方网站提供了测试镜像 Cirros 供用户下载进行功能测试。创建实例前, 用户需要查看 Glance 中的镜像, 并记住需要的镜像 UUID 以便后续创建实例时使用。镜像查看命令如下:

```
[root@controller1 ~]# nova image-list
```

ID	Name	Status	Server
d11f5678-9920-4d1a-908f-65df7f942857	cirros-0.3.4-x86_64	ACTIVE	

```
[root@controller1 ~]# glance image-list
```

ID	Name
d11f5678-9920-4d1a-908f-65df7f942857	cirros-0.3.4-x86_64

(3) 收集安全组信息

安全组控制了用户对实例的访问, 允许访问实例的 IP 地址, 且端口和协议均在安全组

中进行设置。用户可以根据需求创建自己的安全组，并设置相应的访问权限和端口号。安全组查看目录如下：

```
[root@controller1 ~]# nova secgroup-list
```

Id	Name	Description
77b2b5a5-e455-4f38-bc9c-17ccf2be30d9	default	Default security group

```
[root@controller1 ~]# nova secgroup-list-rules default
```

IP Protocol	From Port	To Port	IP Range	Source Group
icmp	-1	-1	0.0.0.0/0	default
				default
tcp	22	22	0.0.0.0/0	

(4) 收集密钥对信息

密钥对用于远程登录实例，对于用户而言，很多外来镜像并不提供登录口令，甚至禁止以密码形式远程 SSH 登录实例，并且只允许通过密钥的形式 SSH 登录。同时，为了安全起见，OpenStack 官方也推荐使用密钥形式 SSH 登录实例。密钥对的查看命令如下：

```
[root@controller1 ~]# nova keypair-list
```

Name	Type	Fingerprint
admin-key	ssh	38:5c:b8:f7:bc:5d:69:2b:ca:74:dd:ca:10:59:ed:c0

(5) 收集网络资源信息

实例创建时需要指定该实例属于哪个网络，以便在创建过程中找到正确的网络及网络固定 IP 等网络资源。网络资源由 Neutron 数目管理控制，查看目录如下：

```
[root@controller1 ~]# neutron net-list
```

id	name
0720855c-f213-4ff2-9d4f-784573b6c12c	ext-net
7eb1df5c-65b5-40d3-a38b-a179dd1f65a5	HA network tenant cbe811690f3c432aa59fbedcf918a793
edc57f9e-a98c-4ef4-94c3-d1fb5d4d9054	admin-net

创建实例所需信息收集完成后，便可通过 Nova boot 命令启动实例。这里将实例启动分为两种方式，即从镜像启动实例和从 Volume 启动实例。从镜像启动实例并不涉及 Volume 挂载操作，而从 Volume 启动实例又分为从镜像启动实例并挂载非引导 Volume 和直接从可引导 Volume 启动实例，后者需要从镜像创建一个可引导的 Volume。

(1) 创建 Image 实例

创建虚拟机最常用的方式就是从镜像启动实例，实例创建后虚拟机系统镜像位于计算节点本地文件系统中（可以是网络文件系统）。要创建一个可正常使用的虚拟机，用户需要提供 flavor、镜像、网络、安全组和 key 等信息。当创建虚拟机所必须的各项资源均已准备完成后，便可按照如下命令语法创建实例（flavor、镜像、网络、key 和安全组需要用用户指定具体值）：

```
$ nova boot --flavor FLAVOR_ID --image IMAGE_ID --key-name KEY_NAME \
--nic net-id NETWORK_ID --user-data USER_DATA_FILE --security-groups\
SEC_GROUP_NAME --file DST-PATH=SRC-PATH --meta KEY=VALUE\
--availability-zone AVAILABILITY_ZONE:COMPUTE_HOSTINSTANCE_NAME
```

在创建实例时，用户可以为实例指定 metadata 键值对，用以对实例进行描述，如 --meta ServerName="My Server"。此外，可以通过 --user-data 参数将包含在本地文件中的用户数据传递到实例中，还可以通过 --file <dst-path=src-path> 选项直接将用户本地文件传递到实例中指定的目录下，一次最多可以传递 5 个文件。--availability-zone 选项可以用来指定创建虚拟机的宿主机，即定向将虚拟机创建到某台特定的物理主机上。如下命令启动一个实例 test-server，使用 flavor 为 m1.tiny（ID 为 1），镜像为 cirros-0.3.4-x86_64，网络为 admin-net，网络 ID 为 edc57f9e-a98c-4ef4-94c3-d1fb5d4d9054，安全组为 default，keypair 为 admin-key，用户数据文件名为 cloudinit.file，metadata 键值对为 ServerName="My Server"，将本地文件 sjx.txt 传递到实例的 /root 目录中：

```
[root@controller1 ~]#nova boot --flavor 1 --image cirros-0.3.4-x86_64\
--key-name admin-key --security-group default --nic \
net-id=edc57f9e-a98c-4ef4-94c3-d1fb5d4d9054 --user-data \
cloudinit.file --file /root=sjx.txt --meta servername="My-server" \
test-server
```

Property	Value
OS-DCF:diskConfig	MANUAL
OS-EXT-AZ:availability_zone	
OS-EXT-SRV-ATTR:host	-
OS-EXT-SRV-ATTR:hostname	test-server
OS-EXT-SRV-ATTR:hypervisor_hostname	-
OS-EXT-SRV-ATTR:instance_name	instance-00000011
OS-EXT-SRV-ATTR:kernel_id	
OS-EXT-SRV-ATTR:launch_index	0
OS-EXT-SRV-ATTR:ramdisk_id	
OS-EXT-SRV-ATTR:reservation_id	r-k4u752xf
OS-EXT-SRV-ATTR:root_device_name	-
OS-EXT-SRV-ATTR:user_dataVkaw5pdC50eHQK
OS-EXT-STS:power_state	0
OS-EXT-STS:task_state	scheduling
OS-EXT-STS:vm_state	building
.....	

实例创建命令返回的信息中, task_state 为 scheduling, vm_state 为 building。这再次说明 Nova-api 将实例创建请求以 RPC cast 的方式传递给 Nova-scheduler 进行调度后, 便直接向 Nova 客户端返回消息, 而 vm_state 为 building, 说明实例创建过程正在后台进行。实例创建完成之后, vm_state 应该为 Active, task_state 为 NOSTATE。检查实例创建过程是否已经正常结束:

```
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State	Networks
...ee61	admin-instance3	ACTIVE	-	Running	admin-net=192.128.1.8
...2f39	test-server	ACTIVE	-	Running	admin-net=192.128.1.12

(2) 创建 Volume 实例

Nova 允许在实例创建的同时将块存储挂载到实例上作为云硬盘使用, 并且允许实例删除后块存储仍然存在, 即保存到云硬盘上的用户数据在虚拟机被销毁后不受影响, 并可以挂载到其他虚拟机上继续使用。此外, Nova 还允许从镜像创建一个可引导的 Volume, 并将实例从此 Volume 上启动, 此时的虚拟机操作系统镜像位于外部块存储上, 因此本地磁盘或文件系统的故障并不影响到虚拟机操作系统的正常使用, 这类操作系统启动方式也称为类 SAN BOOT 启动形式。要创建一个常规不可引导 Volume 并将其挂载到从镜像启动的虚拟机上, 可以按照以下几个步骤操作。

1) 创建常规不可引导 Volume。

```
[root@controller1 ~]# cinder create --display-name volume1 2
```

2) 从镜像启动实例并挂载步骤 1 创建的 Volume。

```
[root@controller1 ~]# nova boot --flavor 1 --image cirros-0.3.4-x86_64 \
--key-name admin-key --security-group default --nic \
net-id=edc57f9e-a98c-4ef4-94c3-d1fb5d4d9054 --block-device \
source=volume,id=0cfd1aef-4667-4ca0-ad66-e5bcbce9a5a2,dest=volume,\
shutdown=preserve test-server
```

此处是从镜像启动 test-server 实例, 同时将步骤 1 创建的块存储 Volume1 挂载到 test-server 上, 注意 --block-device 选项的 shutdown=preserve 设置 (该参数设置告知 Nova 在实例销毁时不要一并删除挂载在其上的块存储)。

3) 查看实例状态。

```
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State	Networks
...

```
|...ee61| admin-instance3 | ACTIVE | - | Running |
admin-net=192.128.1.8 |
|...e02e| test-server | BUILD | block_device_mapping | NOSTATE |
```

```
[root@controller1 ~]# nova list
```

```
| ID | Name | Status | Task State | Power State |
Networks |
```

```
|...ee61| admin-instance3 | ACTIVE | - | Running |
admin-net=192.128.1.8 |
|...e02e| test-server | BUILD | spawning | NOSTATE |
```

```
[root@controller1 ~]# nova list
```

```
| ID | Name | Status | Task State | Power State |
Networks |
```

```
|...ee61| admin-instance3 | ACTIVE | - | Running |
admin-net=192.128.1.8 |
|...e02e| test-server | ACTIVE | - | Running |
admin-net=192.128.1.16|
```

4) 查看 Volume 挂载情况。

```
[root@controller1 ~]# cinder list
```

```
| ID | Status | Name | Size | Volume Type | Bootable | Attached to |
|...a5a2| in-use | volume1 | 2 | - | false | ...3379e02e |
```

上述步骤创建的实例仍然是从镜像启动的虚拟机，仅是在创建实例的同时挂载了 Volume 块存储。要创建具有 SAN BOOT 功能的 Volume，可以按照以下步骤操作：

1) 从镜像创建具有引导功能的 Volume。

```
[root@controller1 ~]# cinder create --image-id=d11f5678-9920-4d1a-908f-
65df7f942857\
--display_name=bootable_volume1 2
```

2) 检查创建的 Volume 引导选项 Bootable 是否为 True。

```
[root@controller1 ~]# cinder list
```

```

+-----+-----+-----+-----+-----+
| ID      | Status | Name      | Size | Volume Type | Bootable |
+-----+-----+-----+-----+-----+
| ...9a5a2 | in-use | volume1   | 2    | -           | false    |
| ...3379e02e |      |           |      |             |          |
| ...009f1 | available | bootable_volume1 | 2    | -           | true     |
+-----+-----+-----+-----+-----+

```

3) 从可引导的 Volume 启动实例。

```

[root@controller1 ~]# nova boot --flavor 1 --key-name admin-key\
--security-group default --nic\
net-id=edc57f9e-a98c-4ef4-94c3-d1fb5d4d9054 --block-device\
source=volume,id=1617463f-e728-4bd2-996a-31fb156009f1,dest=volume,\
size=2,shutdown=preserve,bootindex=0 volume_bootable_server

```

此处创建实例时无须指定 `--image` 参数，镜像在步骤 1 中已经写入 Volume，因此如果镜像较大，步骤 1 会花费较多时间。此处 `--block-device` 选项的 `shutdown=preserve` 参数表示删除虚拟机不影响 Volume；`bootindex=0` 表示此 Volume 为 0 号启动设备，即此虚拟机默认从 Volume 启动；`source=volume` 表示创建块设备的对象类型，除了 `volume` 还可以是 `snapshot`、`image` 和 `blank`；`dest=volume` 表示目标虚拟设备类型，除了 `volume` 还可以是 `local`；`size=2` 表示 `volume` 大小。

4) 查看实例状态。

```
[root@controller1 ~]# nova list
```

```

+-----+-----+-----+-----+-----+
| ID      | Name      | Status | Task State | Power State |
+-----+-----+-----+-----+-----+
| ...ee61 | admin-instance3 | ACTIVE | -          | Running     |
| ...e02e | test-server   | ACTIVE | -          | Running     |
| ...420d | volume_bootable_server | BUILD | block_device_mapping | NOSTATE     |
+-----+-----+-----+-----+-----+

```

[root@controller1 ~]# nova list

```

+-----+-----+-----+-----+-----+
| ID      | Name      | Status | Task State | Power State |
+-----+-----+-----+-----+-----+

```

```

+-----+-----+-----+-----+-----+
|...ee61 | admin-instance3          | ACTIVE | -           | Running  |
|admin-net=192.128.1.8 |
|...e02e | test-server                  | ACTIVE | -           | Running  |
|admin-net=192.128.1.16 |
|...420d | volume_bootable_server      | BUILD  | spawning    | NOSTATE  |
|admin-net=192.128.1.17 |
+-----+-----+-----+-----+-----+

```

```
[root@controller1 ~]# nova list
```

```

+-----+-----+-----+-----+-----+
| ID      | Name                          | Status | Task State | Power State |
| Networks |
+-----+-----+-----+-----+-----+
|...ee61 | admin-instance3          | ACTIVE | -           | Running  |
|admin-net=192.128.1.8 |
|...e02e | test-server              | ACTIVE | -           | Running  |
|admin-net=192.128.1.16 |
|...420d | volume_bootable_server  | ACTIVE | -           | Running  |
|admin-net=192.128.1.17 |
+-----+-----+-----+-----+-----+

```

5) 查看 Volume 状态。

```
[root@controller1 ~]# cinder list
```

```

+-----+-----+-----+-----+-----+-----+-----+
| ID      | Status | Name          | Size | Volume Type | Bootable | Attached to |
+-----+-----+-----+-----+-----+-----+-----+
|...a5a2 | in-use | volume1       | 2    | -           | false    | .....e02e |
|...09f1 | in-use | bootable_volume1 | 2    | -           | true     | .....420d |
+-----+-----+-----+-----+-----+-----+-----+

```

8.7 Nova 实例迁移

8.7.1 Nova 实例 resize/migrate 迁移

Nova 为虚拟机提供了资源升级 (resize) 和主机迁移 (migrate) 操作。从底层调用的 API 接口来看, resize 与 migrate 本质上是相同的, 不同之处在于 resize 需要提供新的资源配置 flavor, 并使用新的 flavor 参数在目标主机上重新启动实例, 而 migrate 则无须 flavor 参数, 或者可以认为当 resize 操作提供的 flavor 与原实例的 flavor 一致时, 则 resize 操作就是 migrate 操作。由于 resize/migrate 操作在迁移过程中会关闭源主机上的实例, 并在新的主机上重新启动实例, 因此 resize/migrate 对实例的迁移并非实时在线, 而是先关闭实例再以 copy 镜像的形式迁移, 迁移完成之后再在新的主机上启动实例, resize/migrate 迁移也称

为“冷迁移”。如果虚拟机系统镜像较大而网络带宽受限时，则 `resize/migrate` 操作可能会花费一定的时间，而且在迁移时间段内不能访问虚拟机。

`resize/migrate` 迁移的优点在于其无须共享存储支持，而且允许用户重新自定义虚拟机资源和宿主机。例如，由于 Nova 调度或者物理设备的原因导致某台宿主机负载过重，而且其上的某些虚拟机比较关键，同时需要更多的物理资源，此时便可申请维护窗口，通过 Nova 的 `resize/migrate` 功能将此宿主机上的某些关键虚拟机迁移到硬件资源更强大的物理宿主主机上。`resize/migrate` 操作无须用户指定目标主机，而是由 Nova-scheduler 调度。`resize` 操作允许源主机与目标主机相同，前提是当前主机的资源满足虚拟机 `resize` 后的资源需求，同时用户需要设置 `nova.conf` 配置文件中的 `allow_resize_to_same_host` 参数为 `true`（默认为 `false`），并重新启动 Nova 相关服务。而 `migrate` 操作则不允许 scheduler 将模板节点选取为源主机，如果 scheduler 调度的目标节点与源节点相同，则会引起 `re-scheduler` 操作（再次调度时源主机机会因 `RetryFilter` 而被排除）。

对于 `resize/migrate` 操作，在迁移过程接近尾声，准备在目标主机启动实例和虚拟机状态为 `resized` 时，还有两个相关操作需要用户执行，即 `confirm_resize` 和 `revert_resize`。`confirm_resize` 操作表示用户接受此次实例迁移操作，`revert_resize` 表示用户不接受（反悔）此次迁移操作。如果用户需要 Nova 自动确认迁移操作，则可以将 `resize_confirm_window` 参数设置为某个大于 0 的时间值，当迁移完成并且虚拟机处于 `resized` 的时间大于此参数值时，迁移操作将会被自动确认。要对虚拟机进行 `resize` 操作，可以按照如下步骤进行（这里允许 `resize` 操作在源主机上进行，因此将 `allow_resize_to_same_host` 设置为 `true`）。

1) 修改计算节点上 `/etc/passwd` 中 `nova` 条目并为用户 `nova` 设置密码。

由于 `resize/migrate` 操作要求计算节点之间实现 SSH 信任无密码互访，而 `nova` 用户默认情况下是禁止登录系统的，因此需要进行 `nova` 用户相关的设置。

```
// /etc/passwd中修改前的nova条目
nova:x:162:162:OpenStack Nova Daemons:/var/lib/nova:/sbin/nologin
// /etc/passwd中修改后的nova条目
nova:x:162:162:OpenStack Nova Daemons:/var/lib/nova:/bin/bash
//设置nova用户的登录密码
[root@compute1 .ssh]# passwd nova
Changing password for user nova.
New password:
BAD PASSWORD: The password is shorter than 8 characters
Retype new password:
passwd: all authentication tokens updated successfully.
```

2) 在各个计算节点上分别生成 SSH 密钥对。

进入 `nova` 用户的 `home` 目录（默认的 `home` 目录为 `/var/lib/nova`）执行 `ssh-keygen` 命令生成密钥对。

```
[root@compute1 ~]# su - nova
-bash-4.2$ ssh-keygen -t rsa
```



```

Generating public/private rsa key pair.
Enter file in which to save the key (/var/lib/nova/.ssh/id_rsa): //回车
Created directory '/var/lib/nova/.ssh'.
Enter passphrase (empty for no passphrase): //回车
Enter same passphrase again: //回车
Your identification has been saved in /var/lib/nova/.ssh/id_rsa.
Your public key has been saved in /var/lib/nova/.ssh/id_rsa.pub.
The key fingerprint is:
3d:91:75:3c:c0:85:90:32:de:01:47:1b:ce:b2:78:b8 nova@compute1
The key's randomart image is:
+--[ RSA 2048 ]-----+
|      .O=+O=O      |
|      oO=O=O      |
|      ..+*      |
|      o.+..      |
|      o S o      |
|      o      |
|      E      |
|      |      |
+-----+
-bash-4.2$ cd .ssh
-bash-4.2$ ll
total 8
-rw----- 1 nova nova 1675 Sep 13 03:31 id_rsa          //私钥
-rw-r--r-- 1 nova nova 395 Sep 13 03:31 id_rsa.pub       //公钥

```

3) 建立计算节点之间的 SSH 信任机制。

resize/migrate 要求计算节点彼此之间可以无密码互访，因此，必须建立计算节点之间的互信机制。SSH 信任机制可以通过在各个计算节点上执行 ssh-copy-id 命令实现。如果不建立计算节点之间 nova 用户的无密码登录，则在 resize/migrate 操作时，源计算节点上将会出现如图 8-53 所示的错误信息，报错原因为无法远程到目标节点上执行 “mkdir” 命令。

```

8a793 - - -] exception during message handling: Bsize error: not able to execute ssh command: unexpected error while running command:
Command: ssh 192.168.142.45 mkdir -p /var/lib/nova/instances/2704539e-f955-4972-bcdd-fb773f96a60
Exit code: 255
Stdout: u
Stderr: u Host key verification failed. r n
2016-09-13 02:59:35.315 51243 ERROR oslo_messaging.rpc.dispatcher Traceback (most recent call last):
2016-09-13 02:59:35.315 51243 ERROR oslo_messaging.rpc.dispatcher File "/usr/lib/python2.7/site-packages/oslo_messaging/rpc/dispatch
reply incoming_message))
2016-09-13 02:59:35.315 51243 ERROR oslo_messaging.rpc.dispatcher File "/usr/lib/python2.7/site-packages/oslo_messaging/rpc/dispatch
return self._do_dispatch(endpoint, method, ctxt, args)
2016-09-13 02:59:35.315 51243 ERROR oslo_messaging.rpc.dispatcher File "/usr/lib/python2.7/site-packages/oslo_messaging/rpc/dispatch
result = func(ctxt, *new_args)
2016-09-13 02:59:35.315 51243 ERROR oslo_messaging.rpc.dispatcher File "/usr/lib/python2.7/site-packages/nova/exception.py", line 11
(payload)
2016-09-13 02:59:35.315 51243 ERROR oslo_messaging.rpc.dispatcher File "/usr/lib/python2.7/site-packages/oslo_utils/excutils.py", 11
self.force_reraise()
2016-09-13 02:59:35.315 51243 ERROR oslo_messaging.rpc.dispatcher File "/usr/lib/python2.7/site-packages/oslo_utils/excutils.py", 11
six.reraise(self.type_, self.value, self.tb)
2016-09-13 02:59:35.315 51243 ERROR oslo_messaging.rpc.dispatcher

```

图 8-53 无 SSH 信任 resize/migrate 迁移报错截图

```

//在compute1节点上执行ssh-copy-id, 目标节点为compute2, 用户为 nova
-bash-4.2$ ssh-copy-id nova@compute2
//ssh无密码登录compute2

```

```

-bash-4.2$ ssh compute2
Last login: Tue Sep 13 03:38:25 2016
-bash-4.2$ hostname
compute2
//在compute2节点上执行ssh-copy-id, 目标节点为compute1, 用户为nova
-bash-4.2$ ssh-copy-id nova@compute1
//ssh无密码登录compute1
-bash-4.2$ ssh compute1
Last login: Tue Sep 13 03:38:45 2016
-bash-4.2$ hostname
compute1

```

//登录成功

4) 新建资源模板。

resize 操作要求提供 flavor 参数, 迁移后的虚拟机资源将由新的 flavor 参数来决定。此处创建一个 ID 为 6, 名称为 m1.tiny_resize, 内存 512M, vCPU 为 1, disk 为 2G, Ephemeral (临时存储设备) 为 1G 的 flavor, 注意默认 flavor 中的 Ephemeral 均为 0。

```

[root@controller1 ~]# nova flavor-create m1.tiny_resize 6 512 2 1 --ephemeral 1
--is-public true

```

```

+-----+
| ID | Name           | Memory_MB | Disk | Ephemeral | Swap | VCPUs |
| RXTX_Factor | Is_Public |
+-----+
| 7 | m2.tiny_resize | 512       | 2    | 1         |      | 1      |
| 1.0 | True         |
+-----+

```

```

[root@controller1 ~]# nova flavor-list

```

```

+-----+
| ID | Name           | Memory_MB | Disk | Ephemeral | Swap | VCPUs |
| RXTX_Factor | Is_Public |
+-----+
| 1 | m1.tiny        | 512       | 1    | 0         |      | 1      |
| 1.0 | True         |
| 2 | m1.small       | 2048      | 20   | 0         |      | 1      |
| 1.0 | True         |
| 3 | m1.medium      | 4096      | 40   | 0         |      | 2      |
| 1.0 | True         |
| 4 | m1.large       | 8192      | 80   | 0         |      | 4      |
| 1.0 | True         |
| 5 | m1.xlarge      | 16384     | 160  | 0         |      | 8      |
| 1.0 | True         |
| 6 | m1.tiny_resize | 512       | 2    | 1         |      | 1      |
| 1.0 | True         |
+-----+

```

5) 开始 resize 迁移。

resize 的对象虚拟机为 admin-instance3, 该虚拟机的原始资源模板 flavor 是 m1.tiny, 其内存为 512M, disk 大小为 1G, 1 个 vCPU, 没有临时存储。另外, admin-instance3 实例在 resize 之前位于 compute1 上。

```
[root@controller1 ~]# nova resize --poll admin-instance3 m1.tiny_resize
```

```
Server resizing... 100% complete
```

```
Finished
```

```
[root@controller1 ~]# nova list
```

```
+-----+-----+-----+-----+-----+
| ID      | Name          | Status      | Task State | Power State |
+-----+-----+-----+-----+-----+
| ...ee61 | admin-instance3 | VERIFY_RESIZE | -          | Running     |
|         | admin-net=192.128.1.8 |             |           |             |
| ...e02e | test-server    | ACTIVE      | -          | Running     |
|         | admin-net=192.128.1.16 |             |           |             |
| ...420d | volume_bootable_server | ACTIVE      | -          | Running     |
|         | admin-net=192.128.1.17 |             |           |             |
+-----+-----+-----+-----+-----+
```

6) 检查 resize 迁移后的实例属性。

由于实例重新由资源模板 m1.tiny_resize 启动, 此时的实例属性中, flavor 值应该为 m1.tiny_resize, 而不是 m1.tiny。此外, 注意 resize 迁移后, admin-instance3 的 hypervisor_hostname 和 vm_state 属性, 这里分别为 compute2 和 resized (不是 active)。

```
[root@controller1 ~]# nova show admin-instance3
```

```
+-----+-----+
| Property                               | Value                               |
+-----+-----+
| OS-DCF:diskConfig                      | MANUAL                             |
| OS-EXT-AZ:availability_zone            | nova                               |
| OS-EXT-SRV-ATTR:host                   | compute2                           |
| OS-EXT-SRV-ATTR:hostname               | admin-instance3                    |
| OS-EXT-SRV-ATTR:hypervisor_hostname   | compute2                         |
| OS-EXT-SRV-ATTR:instance_name          | instance-0000000d                  |
| OS-EXT-SRV-ATTR:kernel_id              |                                     |
| OS-EXT-SRV-ATTR:launch_index           | 0                                   |
| OS-EXT-SRV-ATTR:ramdisk_id             |                                     |
| OS-EXT-SRV-ATTR:reservation_id         | r-tjb573bq                         |
| OS-EXT-SRV-ATTR:root_device_name       | /dev/vda                           |
| OS-EXT-SRV-ATTR:user_data              | -                                   |
| OS-EXT-STS:power_state                  | 1                                   |
| OS-EXT-STS:task_state                   | -                                   |
+-----+-----+
```

OS-EXT-STS:vm_state	resized
OS-SRV-USG:launched_at	2016-09-12T19:45:26.000000
OS-SRV-USG:terminated_at	-
accessIPv4	
accessIPv6	
admin-net network	192.128.1.8
config_drive	
created	2016-09-12T09:35:29Z
description	-
flavor	m1.tiny_resize (6)
hostIdc584b
host_status	UP
id	2704539e-f955-4972-bcd0-fb773f96ee61
image	cirros-0.3.4-x86_64 (....2857)
key_name	admin-key
locked	False
metadata	{}
name	admin-instance3
os-extended-volumes:volumes_attached	[]
progress	0
security_groups	default
status	VERIFY_RESIZE
tenant_id	cbe811690f3c432aa59fbcdcf918a793
updated	2016-09-12T19:45:27Z
user_id	718d52d937a44d26b981296e71dfbe57

注意此时实例的 `vm_state` 一直处于 `VERIFY_RESIZE`，意味着等待用户执行 `resize_confirm` 或 `resize_revert` 操作，即或者确认接受此次虚拟机迁移操作，或者回退到迁移前的状态，如图 8-54 所示。如果是 `resize_confirm` 操作，则删除源节点上的虚拟机镜像，用新的 flavor 启动目标节点上的镜像；如果是回退，则删除目标节点上的镜像，重新启动源节点上的镜像。

云主机

云主机名字 =

筛选

创建云主机

删除云主机

更多操作 ▾

<input type="checkbox"/>	云主机名称	镜像名称	IP 地址	大小	密钥对	状态	可用域	任务	电源状态	从创建以来	动作
<input type="checkbox"/>	volume_bootable_server	-	192.128.1.17	m1.tiny	admin-key	运行	nova	无	运行中	5 小时, 56 分钟	创建快照 ▾
<input type="checkbox"/>	test-server	cirros-0.3.4-x86_64	192.128.1.16	m1.tiny	admin-key	运行	nova	无	运行中	6 小时, 23 分钟	创建快照 ▾
<input type="checkbox"/>	admin-instance3	cirros-0.3.4-x86_64	192.128.1.8	m1.tiny_resize	admin-key	<div>确认或放弃 调整大小/迁移</div>	nova	无	运行中	10 小时, 23 分钟	确认 调整大小/迁移 ▾

正在显示 3 项

图 8-54 resize 迁移后等待用户确认

7) 对 `resize` 迁移执行回退 (`resize_revert`) 操作。

```

+-----+-----+-----+-----+-----+
| ID | Name | Status | Task State | Power State |
+-----+-----+-----+-----+-----+
| ...ee61 | admin-instance3 | ACTIVE | - | Running |
| admin-net=192.128.1.8 |
| ...e02e | test-server | ACTIVE | - | Running |
| admin-net=192.128.1.16 |
| ...420d | volume_bootable_server | ACTIVE | - | Running |
| admin-net=192.128.1.17 |

```

Property	Value
OS-DCF:diskConfig	MANUAL
OS-EXT-AZ:availability_zone	nova
OS-EXT-SRV-ATTR:host	compute1
OS-EXT-SRV-ATTR:hostname	admin-instance3
OS-EXT-SRV-ATTR:hypervisor_hostname	compute1
OS-EXT-SRV-ATTR:instance_name	instance-0000000d
OS-EXT-SRV-ATTR:kernel_id	
OS-EXT-SRV-ATTR:launch_index	0
OS-EXT-SRV-ATTR:ramdisk_id	
OS-EXT-SRV-ATTR:reservation_id	r-tjb573bq
OS-EXT-SRV-ATTR:root_device_name	/dev/vda
OS-EXT-SRV-ATTR:user_data	-
OS-EXT-STS:power_state	1
OS-EXT-STS:task_state	-
OS-EXT-STS:vm_state	active
OS-SRV-USG:launched_at	2016-09-12T20:03:33.000000
OS-SRV-USG:terminated_at	-
accessIPv4	
accessIPv6	
admin-net network	192.128.1.8
config_drive	
created	2016-09-12T09:35:29Z
description	-
flavor	m1.tiny (1)
hostId4b144742b
host_status	UP
id	2704539e-f955-4972-bcd0-fb773f96ee61
image	cirros-0.3.4-x86_64 (.....42857)
key_name	admin-key
locked	False


```

| metadata | {} |
| name | admin-instance3 |
| os-extended-volumes:volumes_attached | [] |
| progress | 0 |
| security_groups | default |
| status | ACTIVE |
| tenant_id | cbe811690f3c432aa59fbedcf918a793 |
| updated | 2016-09-12T20:03:36Z |
| user_id | 718d52d937a44d26b981296e71dfbe57 |
+-----+

```

8) 对 resize 迁移执行确认 (resize_confirm) 操作。

```

[root@controller1 ~]# nova resize-confirm admin-instance3
[root@controller1 ~]# nova list

```

```

+-----+-----+-----+-----+-----+
| ID | Name | Status | Task State | Power State |
+-----+-----+-----+-----+-----+
| ...ee61 | admin-instance3 | ACTIVE | - | Running |
| admin-net=192.128.1.8 |
| ...e02e | test-server | ACTIVE | - | Running |
| admin-net=192.128.1.16 |
| ...420d | volume_bootable_server | ACTIVE | - | Running |
| admin-net=192.128.1.17 |
+-----+-----+-----+-----+-----+

```

```

[root@controller1 ~]# nova show admin-instance3

```

```

+-----+-----+
| Property | Value |
+-----+-----+
| OS-DCF:diskConfig | MANUAL |
| OS-EXT-AZ:availability_zone | nova |
| OS-EXT-SRV-ATTR:host | compute2 |
| OS-EXT-SRV-ATTR:hostname | admin-instance3 |
| OS-EXT-SRV-ATTR:hypervisor_hostname | compute2 |
| OS-EXT-SRV-ATTR:instance_name | instance-0000000d |
| OS-EXT-SRV-ATTR:kernel_id | |
| OS-EXT-SRV-ATTR:launch_index | 0 |
| OS-EXT-SRV-ATTR:ramdisk_id | |
| OS-EXT-SRV-ATTR:reservation_id | r-tjb573bq |
| OS-EXT-SRV-ATTR:root_device_name | /dev/vda |
| OS-EXT-SRV-ATTR:user_data | - |
| OS-EXT-STS:power_state | 1 |
| OS-EXT-STS:task_state | - |
| OS-EXT-STS:vm_state | active |
| OS-SRV-USG:launched_at | 2016-09-12T20:05:33.000000 |
| OS-SRV-USG:terminated_at | - |
| accessIPv4 | |

```

```
| accessIPv6 | |
| admin-net network | 192.128.1.8 |
| config_drive | |
| created | 2016-09-12T09:35:29Z |
| description | - |
| flavor | m1.tiny_resize (6) |
| hostId | .....6faadcc584b |
| host_status | UP |
| id | 2704539e-f955-4972-bcd0-fb773f96ee61 |
| image | cirros-0.3.4-x86_64 (.....942857) |
| key_name | admin-key |
| locked | False |
| metadata | {} |
| name | admin-instance3 |
| os-extended-volumes:volumes_attached | [] |
| progress | 0 |
| security_groups | default |
| status | ACTIVE |
| tenant_id | cbe811690f3c432aa59fbedcf918a793 |
| updated | 2016-09-12T20:06:48Z |
| user_id | 718d52d937a44d26b981296e71dfbe57 |
+-----+-----+
```

当用户执行 `resize_confirm` 操作后，原实例镜像将被删除，而且再也不能回退。确认后，虚拟机的资源将按照 `m1.tiny_resize` 模板来分配，如图 8-55 所示。由于新模板 `m1.tiny_resize` 的 `disk` 为 2G，并且临时存储设备 `Ephemeral` 为 1G，因此 `admin_instance3` 实例在 `resize` 迁移后的 `/dev/vda` 为 2GB，并且新增了 1GB 的临时存储设备 `/dev/vdb`。而在 `m1.tiny` 模板中，`/dev/vda` 为 1GB，没有 `/dev/vdb` 设备。

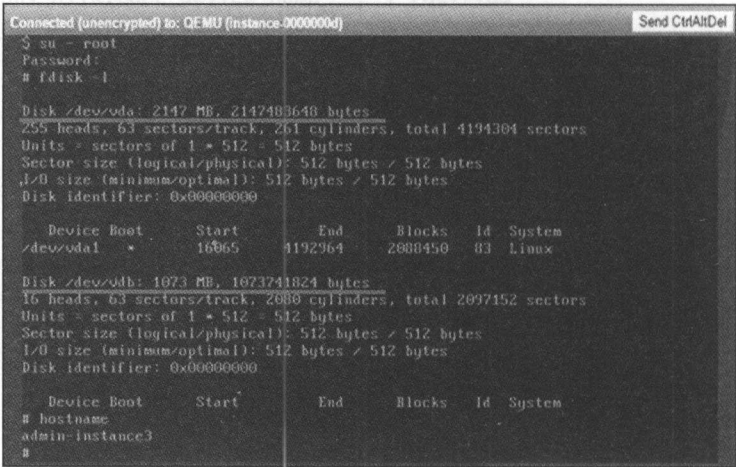


图 8-55 `resize` 操作后虚拟机磁盘资源变化

如果不需要进行虚拟机资源调整，只需进行纯粹的“冷迁移”`migrate`操作即可。

migrate 与 resize 类似，但是 migrate 操作无须指定 flavor 参数，仅仅是将实例迁移至其他主机，而不进行资源调整。migrate 操作步骤如下。

1) 迁移之前检查迁移对象虚拟机属性。

```
[root@controller1 ~]# nova show admin-instance3
```

Property	Value
OS-DCF:diskConfig	MANUAL
OS-EXT-AZ:availability_zone	nova
OS-EXT-SRV-ATTR:host	compute2
OS-EXT-SRV-ATTR:hostname	admin-instance3
OS-EXT-SRV-ATTR:hypervisor_hostname	compute2
OS-EXT-SRV-ATTR:instance_name	instance-0000000d
OS-EXT-SRV-ATTR:kernel_id	
OS-EXT-SRV-ATTR:launch_index	0
OS-EXT-SRV-ATTR:ramdisk_id	
OS-EXT-SRV-ATTR:reservation_id	r-tjb573bq
OS-EXT-SRV-ATTR:root_device_name	/dev/vda
OS-EXT-SRV-ATTR:user_data	-
OS-EXT-STS:power_state	1
OS-EXT-STS:task_state	-
OS-EXT-STS:vm_state	active
OS-SRV-USG:launched_at	2016-09-12T20:05:33.000000
OS-SRV-USG:terminated_at	-
accessIPv4	
accessIPv6	
admin-net network	192.128.1.8
config_drive	
created	2016-09-12T09:35:29Z
description	-
flavor	m1.tiny_resize (6)
hostIdf6faadcc584b
host_status	UP
id	2704539e-f955-4972-bcd0-fb773f96ee61
image	cirros-0.3.4-x86_64 (.....7f942857)
key_name	admin-key
locked	False
metadata	{}
name	admin-instance3
os-extended-volumes:volumes_attached	[]
progress	0
security_groups	default
status	ACTIVE
tenant_id	cbe811690f3c432aa59fbcdcf918a793
updated	2016-09-12T20:06:48Z
user_id	718d52d937a44d26b981296e71dfbe57

2) 开始 migrate 迁移。

```
[root@controller1 ~]# nova migrate --poll admin-instance3
```

```
Server migrating... 100% complete
Finished
```

3) 检查 migrate 后的实例状态。

```
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State
Networks				
...ee61	admin-instance3	VERIFY_RESIZE	-	Running
admin-net=192.128.1.8				
...e02e	test-server	ACTIVE	-	Running
admin-net=192.128.1.16				
...420d	volume_bootable_server	ACTIVE	-	Running
admin-net=192.128.1.17				

```
[root@controller1 ~]# nova show admin-instance3
```

Property	Value
OS-DCF:diskConfig	MANUAL
OS-EXT-AZ:availability_zone	nova
OS-EXT-SRV-ATTR:host	compute1
OS-EXT-SRV-ATTR:hostname	admin-instance3
OS-EXT-SRV-ATTR:hypervisor_hostname	compute1
OS-EXT-SRV-ATTR:instance_name	instance-0000000d
OS-EXT-SRV-ATTR:kernel_id	
OS-EXT-SRV-ATTR:launch_index	0
OS-EXT-SRV-ATTR:ramdisk_id	
OS-EXT-SRV-ATTR:reservation_id	r-tjb573bq
OS-EXT-SRV-ATTR:root_device_name	/dev/vda
OS-EXT-SRV-ATTR:user_data	-
OS-EXT-STS:power_state	1
OS-EXT-STS:task_state	-
OS-EXT-STS:vm_state	resized
OS-SRV-USG:launched_at	2016-09-12T20:14:32.000000
OS-SRV-USG:terminated_at	-
accessIPv4	
accessIPv6	
admin-net network	192.128.1.8
config_drive	
created	2016-09-12T09:35:29Z
description	-
flavor	m1.tiny_resize (6)
hostIdfla4b144742b
host_status	UP
id	2704539e-f955-4972-bcd0-fb773f96ee61

```

| image | cirros-0.3.4-x86_64 (.....f942857) |
| key_name | admin-key |
| locked | False |
| metadata | {} |
| name | admin-instance3 |
| os-extended-volumes:volumes_attached | [] |
| progress | 0 |
| security_groups | default |
| status | VERIFY_RESIZE |
| tenant_id | cbe811690f3c432aa59fbedcf918a793 |
| updated | 2016-09-12T20:14:33Z |
| user_id | 718d52d937a44d26b981296e71dfbe57 |
+-----+

```

4) 执行 migrate 的确认 (confirm) 操作。

```

[root@controller1 ~]# nova resize-confirm admin-instance3
[root@controller1 ~]# nova list
+-----+-----+-----+-----+-----+
| ID | Name | Status | Task State | Power State |
| Networks |
+-----+-----+-----+-----+-----+
| ...ee61 | admin-instance3 | ACTIVE | - | Running |
| admin-net=192.128.1.8 |
| ...e02e | test-server | ACTIVE | - | Running |
| admin-net=192.128.1.16 |
| ...420d | volume_bootable_server | ACTIVE | - | Running |
| admin-net=192.128.1.17 |

```

8.7.2 Nova 实例 live-migration 迁移

为了进行服务非中断的实时维护，Nova 提供了 live-migration 功能。相对于 resize/migrate 的“冷迁移”，live-migration 称为“热迁移”，即实时在线迁移，其可以将实例由一个计算节点在线迁移到另一个计算节点，期间仅有非常短暂的访问延时。live-migration 迁移按其实现方式可以划分为三种类型，即基于非共享存储的块迁移 (block live migration)、基于共享存储的迁移 (Shared storage-based live migration) 和基于 Volume 后端的迁移 (Volume-backed live migration)。其中，后两种是使用最多的 live-migration 方式，而块迁移在使用上一直存在很多问题，而且也不符合实时迁移的基本设计思想，故不推荐使用。块迁移在迁移过程中需要拷贝临时存储及镜像文件，对于节点之间的网络带宽要求极高，尤其是在临时存储很大的情况下，还有可能出现消耗很长时间却不能成功的情况，如当实例 I/O 负载很大时，如果应用程序对临时存储的写入速率快于块迁移对临时存储的拷贝速率，则块迁移将不能完成。基于上述原因，本节仅介绍基于共享存储和基于 Volume 后端的实例在线迁移。

使用 live-migration 功能时，需要对 Nova 和 Libvirt 做相应的修改，而不管是基于共

享存储还是 Volume 的迁移, 迁移前的准备工作都是一样的, 即可以实现共享存储的 live-migration, 就可以实现基于 Volume 的 live-migration。要实现 live-migration 功能, 需要在每个计算节点上按如下步骤进行迁移前的准备工作。

1) 修改 Nova 配置文件 `/etc/nova/nova.conf`。如果是 Mitaka 版本, 则无须更改, 否则 VNC 功能不可用。

```
vncserver_proxyclient_address=127.0.0.1
vncserver_listen=0.0.0.0
//live_migration_flag是个即将抛弃的参数, 可以不用设置
live_migration_flag=VIR_MIGRATE_UNDEFINE_SOURCE,VIR_MIGRATE_PEER2PEER
```

2) 修改 Libvirt 配置文件 `/etc/libvirt/libvirtd.conf`。

```
//修改前
.....
#listen_tls = 0
#listen_tcp = 1
#auth_tcp = "sasl"
.....
//修改后
.....
listen_tls = 0
listen_tcp = 1
auth_tcp="none"
.....
```

3) 修改配置文件 `/etc/sysconfig/libvirtd?`。

```
//修改前
.....
#LIBVIRT_ARGS="--listen"
.....
//修改后
.....
LIBVIRT_ARGS="--listen"
.....
```

4) 重启 Nova 服务和 libvirtd 进程。

```
systemctl restart openstack-nova-compute
systemctl restart libvirtd
```

live-migration 准备工作完成之后, 便可开始 live-migration 迁移, 此处介绍基于 Volume 后端的实例迁移和基于 NFS 共享存储的实例迁移。基于 Volume 后端的实例迁移其实就是对 SAN BOOT 形式的实例进行迁移, 由于实例系统位于 Volume 而非临时磁盘上, 因此无须共享存储, 其迁移过程就是将 Volume 从源主机卸载, 并重新挂载到目标主机的过程。

1. 基于 Volume 后端的 live-migration

要实现基于 Volume 后端的实例迁移, 按照如下步骤操作即可:

1) 迁移前检查对象实例位于哪个 Hypervisor 主机。此处迁移的实例名称为 volume_bootable_server (UUID 的末四位是 420d), 这是一个基于 Volume 后端的实例, 对应的 Volume 名称为 bootable_volume1。

```
[root@controller1 ~]# nova list
```

```
+-----+-----+-----+-----+-----+
| ID | Name | Status | Task State | Power State |
+-----+-----+-----+-----+-----+
| ...ee61 | admin-instance3 | ACTIVE | - | Running |
| admin-net=192.128.1.8 |
| ...e02e | test-server | ACTIVE | - | Running |
| admin-net=192.128.1.16 |
| ...420d | volume_bootable_server | ACTIVE | - | Running |
| admin-net=192.128.1.17 |
```

```
[root@controller1 ~]# nova hypervisor-servers compute1
```

```
+-----+-----+-----+-----+-----+
| ID | Name | Hypervisor ID | Hypervisor Hostname |
+-----+-----+-----+-----+-----+
| ...420d | instance-00000016 | 1 | compute1 |
| ...ee61 | instance-0000000d | 1 | compute1 |
```

```
[root@controller1 ~]# nova hypervisor-servers compute2
```

```
+-----+-----+-----+-----+-----+
| ID | Name | Hypervisor ID | Hypervisor Hostname |
+-----+-----+-----+-----+-----+
| ...e02e | instance-00000014 | 2 | compute2 |
```

```
[root@controller1 ~]# cinder list
```

```
+-----+-----+-----+-----+-----+-----+-----+
| ID | Status | Name | Size | Volume Type | Bootable | Attached to |
+-----+-----+-----+-----+-----+-----+-----+
| ...a5a2 | in-use | volume1 | 2 | - | false | ...e02e |
| ...09f1 | in-use | bootable_volume1 | 2 | - | true | ...420d |
```

2) 检查 Hypervisor 主机的剩余可用资源。

```
//源Hypervisor主机
```

```
[root@controller1 ~]# nova hypervisor-show compute1
```

```
+-----+-----+
| Property | Value |
+-----+-----+
| ..... | ..... |
| current_workload | 0 |
| disk_available_least | 5 |
```

```

| free_disk_gb      | 11
| free_ram_mb       | 800
| host_ip           | 192.168.142.44
| hypervisor_hostname | compute1
| hypervisor_type    | QEMU
| hypervisor_version | 1005003
| id                | 1
| local_gb          | 12
| local_gb_used      | 1
| memory_mb         | 1824
| memory_mb_used     | 1024
| running_vms       | 1
| service_disabled_reason | None
| service_host      | compute1
| service_id        | 11
| state             | up
| status            | enabled
| vcpus             | 2
| vcpus_used        | 1

```

//目标Hypervisor主机

```
[root@controller1 ~]# nova hypervisor-show compute2
```

```

| Property          | Value
| .....
| current_workload   | 0
| disk_available_least | 4
| free_disk_gb       | 10
| free_ram_mb        | 288
| host_ip            | 192.168.142.45
| hypervisor_hostname | compute2
| hypervisor_type     | QEMU
| hypervisor_version  | 1005003
| id                 | 2
| local_gb           | 12
| local_gb_used       | 2
| memory_mb          | 1824
| memory_mb_used      | 1536
| running_vms        | 2
| service_disabled_reason | None
| service_host       | compute2
| service_id         | 12
| state              | up
| status             | enabled
| vcpus              | 2
| vcpus_used         | 2

```

3) 开始进行 live-migration 迁移。

```
[root@controller1 ~]# nova live-migration volume_bootable_server \
compute2
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State
...ee61	admin-instance3	ACTIVE	-	Running
...e02e	test-server	ACTIVE	-	Running
...420d	volume_bootable_server	MIGRATING	migrating	Running

```
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State
...ee61	admin-instance3	ACTIVE	-	Running
...e02e	test-server	ACTIVE	-	Running
...420d	volume_bootable_server	ACTIVE	-	Running

4) 迁移完成后检查实例 Hypervisor 主机是否改变。

```
[root@controller1 ~]# nova hypervisor-servers compute1
```

ID	Name	Hypervisor ID	Hypervisor Hostname
...ee61	instance-0000000d	1	compute1

// volume_bootable_server的hypervisor 主机已经变为compute2

```
[root@controller1 ~]# nova hypervisor-servers compute2
```

ID	Name	Hypervisor ID	Hypervisor Hostname
...e02e	instance-00000014	2	compute2
...420d	instance-00000016	2	compute2

2. 基于 NFS 共享存储的 live-migration 迁移

与基于 Volume 后端的 live-migration 不同，基于 NFS 的迁移需要计算节点之间共享实

例镜像文件目录，通常是 /var/lib/nova/instances，即每个计算节点都可以对共享的 /var/lib/nova/instances 目录进行读写。为了便于演示，这里将控制节点 controller1 作为 NFS 的服务器以导出共享目录，计算节点作为 NFS 的客户端挂载共享目录到 /var/lib/nova/instances，并在共享目录中创建实例 nfs-server 用作 live-migration 迁移对象。为了说明 live-migration 迁移会自动重新挂载实例的 Volume，在迁移前为实例 nfs-server 挂载一块大小 1GB 的 Volume。基于 NFS 的 live-migration 操作步骤如下。

1) 配置控制节点为 NFS 服务器并导出共享目录。

```
[root@controller1 ~]# mkdir -p /nova/instances
[root@controller1 ~]# chown -R nova:nova /nova/instances
[root@controller1 ~]# echo "/nova/instances \
192.168.142.0/24(rw,sync,no_root_squash)">>/etc/exports
[root@controller1 ~]# exportfs -r
[root@controller1 ~]# exportfs -a
[root@controller1 ~]# showmount -e
Export list for controller1:
/etc/yum.repos.d *
/data *
/nova/instances 192.168.142.0/24 //192.168.142.0/24为计算节点网络
```

2) 计算节点挂载 NFS 目录到 /var/lib/nova/instances。

```
//计算节点compute1
[root@compute1 nova]# mount controller1:/nova/instances /var/lib/nova/instances
[root@compute1 nova]# df -h
Filesystem                Size  Used Avail Use% Mounted on
.....
controller1:/nova/instances    13G  7.0G  6.0G  54% /var/lib/nova/instances
[root@compute1 instances]# ls -l /var/lib/nova/instances
total 0
//计算节点compute2
[root@compute2 ~]# mount controller1:/nova/instances /var/lib/nova/instances
[root@compute2 ~]# df -h
Filesystem                Size  Used Avail Use% Mounted on
.....
controller1:/nova/instances    13G  7.0G  6.0G  54% /var/lib/nova/instances
[root@compute2 ~]# ls -l /var/lib/nova/instances
total 0
```

3) 创建基于 NFS 共享目录的实例并挂载 Volume，挂载 Volume 后实例系统中的块存储如图 8-56 所示。

```
[root@controller1 ~]# nova boot --flavor 1 --image cirros-0.3.4-x86_64\
--key-name admin-key --security-group default --nic \
net-id=edc57f9e-a98c-4ef4-94c3-d1fb5d4d9054 nfs_server
//nfs-server位于compute1上
[root@controller1 ~]# nova show nfs_server
```



```

| Property | Value |
+-----+-----+
| OS-DCF:diskConfig | MANUAL |
| OS-EXT-AZ:availability_zone | nova |
| OS-EXT-SRV-ATTR:host | compute1 |
| OS-EXT-SRV-ATTR:hostname | nfs-server |
| OS-EXT-SRV-ATTR:hypervisor_hostname | compute1 |
.....
//创建volume
[root@controller1 ~]# cinder create --display-name volume1 1
//挂载volume
[root@controller1 ~]# nova volume-attach 3be393f3-04f6-4437-b0e3-c2cbf873d4e0\
fe8c7aaf-3bc8-41e4-9623-0ba506030894 auto
+-----+-----+
| Property | Value |
+-----+-----+
| device | /dev/vdb |
| id | fe8c7aaf-3bc8-41e4-9623-0ba506030894 |
| serverId | 3be393f3-04f6-4437-b0e3-c2cbf873d4e0 |
| volumeId | fe8c7aaf-3bc8-41e4-9623-0ba506030894 |
+-----+-----+
[root@controller1 ~]# cinder list
+-----+-----+-----+-----+-----+-----+-----+
| ID | Status | Name | Size | Volume Type | Bootable | Attached to |
+-----+-----+-----+-----+-----+-----+-----+
| ...0894 | in-use | volume1 | 1 | - | false | ...873d4e0 |
+-----+-----+-----+-----+-----+-----+-----+

```

```

$ su - root
Password:
# fdisk -l

Disk /dev/vda: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders, total 2097152 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
/dev/vda1   *        16065       2088449       1036192   83   Linux

Disk /dev/vdb: 1073 MB, 1073741824 bytes
16 heads, 63 sectors/track, 2080 cylinders, total 2097152 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

Disk /dev/vdb doesn't contain a valid partition table
# hostname
nfs-server

```

图 8-56 迁移前实例 nfs-server 中的块存储

4) 查看每个计算节点的 /var/lib/nova/instances 目录, 正常情况下该目录内容应该完全相同, 为 nfs-server 实例 (UUID 末尾 4 位为 d4e0) 的镜像相关文件。

```

//控制节点上(nfs server)
[root@controller1 ~]# ls -l /nova/instances

```

```
total 0
drwxr-xr-x 2 nova nova 69 Sep 13 11:38 3be393f3-04f6-4437-b0e3-c2cbf873d4e0
drwxr-xr-x 2 nova nova 53 Sep 13 11:38 _base
drwxr-xr-x 2 nova nova 58 Sep 13 11:38 locks
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State
Networks				
...	d4e0 nfs_server	ACTIVE	-	Running
	admin-net=192.128.1.19			
...	e02e test-server	SHUTOFF	-	Shutdown
	admin-net=192.128.1.16			
...	420d volume_bootable_server	SHUTOFF	-	Shutdown
	admin-net=192.128.1.17			

```
//计算节点compute1上
[root@compute1]# ls -l /var/lib/nova/instances
total 0
```

```
drwxr-xr-x 2 nova nova 69 Sep 13 11:38 3be393f3-04f6-4437-b0e3-c2cbf873d4e0
drwxr-xr-x 2 nova nova 53 Sep 13 11:38 _base
drwxr-xr-x 2 nova nova 58 Sep 13 11:38 locks
```

```
//计算节点compute2上
[root@compute2 ~]# ls -l /var/lib/nova/instances
total 0
```

```
drwxr-xr-x 2 nova nova 69 Sep 13 11:38 3be393f3-04f6-4437-b0e3-c2cbf873d4e0
drwxr-xr-x 2 nova nova 53 Sep 13 11:38 _base
drwxr-xr-x 2 nova nova 58 Sep 13 11:38 locks
```

5) 在 live-migration 迁移前检查 Hypervisor 主机情况。

```
[root@controller1 ~]# nova hypervisor-servers compute1
```

ID	Name	Hypervisor ID
Hypervisor Hostname		
3be393f3-04f6-4437-b0e3-c2cbf873d4e0	instance-0000001a	1
compute1		
0758d089-54f7-4182-a749-54e98ac7420d	instance-00000016	1
compute1		

```
[root@controller1 ~]# nova hypervisor-servers compute2
```

ID	Name	Hypervisor ID
----	------	---------------

```

Hypervisor Hostname |
+-----+
+-----+
| 880fac91-9c76-4452-8508-36d73379e02e | instance-00000014 | 2 |
| compute2 |
+-----+
+-----+
[root@controller1 ~]# nova list
+-----+
+-----+
| ID | Name | Status | Task State | Power State |
| Networks |
+-----+
+-----+
| ...d4e0 | nfs_server | ACTIVE | - | Running |
| admin-net=192.128.1.19 |
| ...e02e | test-server | SHUTOFF | - | Shutdown |
| admin-net=192.128.1.16 |
| ...420d | volume_bootable_server | SHUTOFF | - | Shutdown |
| admin-net=192.128.1.17 |
+-----+
+-----+

```

6) 开始 live-migration 迁移。

```

[root@controller1 ~]# nova live-migration nfs_server compute2
[root@controller1 ~]# nova list
+-----+
+-----+
| ID | Name | Status | Task State | Power State |
| Networks |
+-----+
+-----+
| ...d4e0 | nfs_server | MIGRATING | migrating | Running |
| admin-net=192.128.1.19 |
| ...e02e | test-server | SHUTOFF | - | Shutdown |
| admin-net=192.128.1.16 |
| ...420d | volume_bootable_server | SHUTOFF | - | Shutdown |
| admin-net=192.128.1.17 |
+-----+
+-----+
[root@controller1 ~]# nova list
+-----+
+-----+
| ID | Name | Status | Task State | Power State |
| Networks |
+-----+
+-----+
| ...d4e0 | nfs_server | ACTIVE | - | Running |
| admin-net=192.128.1.19 |

```

```
|...e02e | test-server | SHUTOFF | - | Shutdown |
admin-net=192.128.1.16|
|...420d | volume_bootable_server | SHUTOFF | - | Shutdown |
admin-net=192.128.1.17|
+-----+
-----+
```

7) live-migration 迁移完成后，检查实例 Hypervisor 主机是否改变。

```
[root@controller1 ~]# nova hypervisor-servers compute2
+-----+
-----+
| ID | Name | Hypervisor ID |
Hypervisor Hostname |
+-----+
+-----+
| 880fac91-9c76-4452-8508-36d73379e02e | instance-00000014 | 2 |
compute2 |
| 3be393f3-04f6-4437-b0e3-c2cbf873d4e0 | instance-0000001a | 2 |
compute2 |
+-----+
-----+
[root@controller1 ~]# nova hypervisor-servers compute1
+-----+
-----+
| ID | Name | Hypervisor ID |
Hypervisor Hostname |
+-----+
+-----+
| 0758d089-54f7-4182-a749-54e98ac7420d | instance-00000016 | 1 |
compute1 |
+-----+
-----+
```

8) live-migration 迁移完成后，检查块存储 Volume 是否重新挂载并能正常使用，如图 8-57 所示。

```
[root@controller1 ~]# cinder list
+-----+
+-----+
| ID | Status | Name | Size | Volume Type | Bootable | Attached to |
+-----+
|...0894 | in-use | volume1 | 1 | - | false | ...873d4e0 |
+-----+
```

Nova 在配置文件 nova.conf 中提供了关于 live-migration 功能的多个配置参数，通常情况下这些参数都有默认值，并且无须做任何改动即可支持 live-migration 功能的正常使用。当然，用户可以根据自己的实际需求对 live-migration 参数进行修改，表 8-13 是 live-migration 各个参数的功能解释及其默认值列表。

```
# fdisk -l
Disk /dev/vda: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders, total 2097152 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
/dev/vda1 *        16065       2088449     1036192+   83   Linux

Disk /dev/vdb: 1073 MB, 1073741824 bytes
16 heads, 63 sectors/track, 2080 cylinders, total 2097152 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

Disk /dev/vdb doesn't contain a valid partition table
# hostname
nfs-server
```

图 8-57 迁移后 nfs-server 中的块存储

表 8-13 live-migration 参数配置选项

默认值	描 述
live_migration_retry_count = 30	整数, live_migration 操作每秒钟尝试次数
max_concurrent_live_migrations = 1	整数, 允许并行执行的最大 live-migration 操作数
[libvirt]	
live_migration_bandwidth = 0	整数, 允许 live-migration 使用的最大带宽 (MiB/s), 如果为 0, live-migration 操作为自动选择合适的默认值
live_migration_completion_timeout = 800	整数, live-migration 迁移等待时间 (单位为秒), 允许在此时间段内迁移, 超时将停止迁移操作。
live_migration_downtime = 500	整数, 允许 live-migration 操作中实例切换时的最大 down 机时间 (单位为毫秒)
live_migration_downtime_delay = 75	整数, live-migration 过程中两个增量步之间的等待时间间隔
live_migration_downtime_steps = 10	整数, 最大增量步数, 默认 10, 最小 3
live_migration_flag = VIR_MIGRATE_UNDEFINE_SOURCE, VIR_MIGRATE_PEER2PEER, VIR_MIGRATE_LIVE, VIR_MIGRATE_TUNNELLED	字符串, live-migration 迁移标志, 正确的 live-migration 迁移标志可以从 live_migration_tunnelled 配置项推导出来, 此参数已被标记为抛弃, 以后版本中不再使用
live_migration_inbound_addr = None	字符串, live-migration 的目标 IP 或主机名, 如果为 none, 则使用迁移时指定的目标计算节点主机名
live_migration_progress_timeout = 150	整数, 终止 live-migration 操作之前等待传输数据的时间 (单位为秒)
live_migration_tunnelled = None	布尔值, 是否使用 tunnelled 迁移。如果为 true, 则使用 VIR_MIGRATE_TUNNELLED 标志进行迁移, 迁移数据基于 libvirtd 连接传输, 如果为 false, 使用 native 传输, 如果未设置, Nova 自动选择默认传输方式
live_migration_uri = None	字符串, 用于覆盖 libvirt live-migration 的目标 URL, 字符串值根据 virt_type 值而定

8.8 Nova 实例高可用

8.8.1 Nova 实例高可用概述

在云计算环境，尤其是私有云环境中，虚拟机的高可用性一直是阻碍企业进行私有云建设的痛点，对于 OpenStack 更是如此。由于 OpenStack 的初衷是面向公有云，因此 OpenStack 在“基因”里便缺乏实例高可用设计，这也是到目前为止 OpenStack 的计算服务 Nova 一直没有完善的实例高可用功能的原因。在公有云环境中，用户业务系统的高可用性应该更多地依赖于分布在集群中的应用程序本身，即公有云上运行的应用程序有自己的集群和负载均衡系统，能在一定程度上容忍物理计算节点宕机所带来的实例不可用，并能通过负载均衡系统实现故障计算节点上的负载自动转移。而随着越来越多传统 IT 架构下的用户转入云计算，OpenStack 的私有云用户不断增加，由于传统集中式应用系统的非中断运行几乎完全依赖于服务器的高可用性，因此在私有云环境下，OpenStack Nova 服务的虚拟机实例高可用性变得极为迫切，也是众多 OpenStack 私有云用户所急需的功能。然而，实现虚拟机高可用并非 OpenStack 或者云计算的核心任务，不能为了照顾传统应用系统而使得代表新一代 IT 革命的云计算走上回头之路。在新的 IT 时代，用户应该抛弃传统应用系统或者对其进行革新以适应云计算环境，而不是倒逼云计算去适应传统应用系统。换句话说，传统的“宠物”式服务器聚焦维护方式必然被淘汰，而对服务器持以“绵羊”式的放养心态才是未来 IT 的发展方向，这也是云计算环境中对待服务器的核心态度。这一观点，也正好能很好地解释为何 OpenStack 社区，尤其是 Nova 团队，长期以来不致力于（甚至是拒绝接受类似的功能项目）为 Nova 的虚拟机实例实现完善高可用功能的原因。

但是，对于公众而言，云计算仍然还处于发展阶段，仍然面临着传统 IT 架构到云计算架构的过渡。尽管 OpenStack 几乎统一了云计算的 IaaS 层，但是到目前为止也仅有 7 年时间，而且部署 OpenStack 私有云的用户也日益增加，为了兼顾和确保传统应用系统平滑向云计算过渡，虽然 OpenStack 社区并没有提供完整的实例高可用解决方案，但也提供了部分配合外部监控服务的工作机制以使得用户可以自己实现实例高可用方案。此外，在 Liberty 版本中，OpenStack 还实现、改进了相关的 NovaAPI 接口，以便更好地配合外部高可用系统实现对 Nova 服务状态改变的监控和对虚拟机进行故障转移。基于这些服务，各个 OpenStack 厂商也都推出了自己的 OpenStack 计算服务高可用方案，以帮助用户实现虚拟机实例的高可用。例如红帽的 RDO 便实现了基于 Pacemaker-remote 的实例高可用方案，同时社区也有基于 Zookeeper 对 nova-compute 服务进行监控以实现实例高可用的方案。

在 OpenStack 中，所谓计算节点实例高可用，是指在物理计算节点发生硬件故障时（如磁盘损坏、CPU 或内存故障导致宕机、物理网络故障和电源故障），自动将该节点关闭，并让其上的虚拟机在计算节点集群中的其余健康节点上重启，如果还能实现实例的动态迁移，便是最佳的高可用方案。在实现 OpenStack 实例高可用的方案中，虽然各种方案所使用的软件不同，但是通常都是基于三个步骤来实现，即监控（Monitoring）、隔离（Fencing）和

恢复 (Recovery)。

(1) 监控

监控的目的在于判断 Hypervisor 是否故障, 并为隔离操作提供执行依据。监控功能由两部分构成, 一是检测主机故障情况, 二是触发主机故障后的自动响应任务 (隔离和恢复)。关于监控功能是否应该集成到 Nova 中, 社区一直存在争论。其中, 认为计算节点服务的监控应该集成到 Nova 项目中, 主要是基于 Nova 服务一定程度上已能获取其自身运行所依赖基础架构的健康状况, 或者说 Nova 本身已经可以跟踪活动的计算节点。但是, Nova 对计算节点的跟踪监控只能探测 Nova-compute 服务的故障与否, 而 Nova-compute 服务的故障并不意味着虚拟机实例也出现故障, 即计算节点 Nova-compute 服务正常与否与其上的虚拟机故障与否并无必然联系。若将对基础架构的监控集成到 Nova 中, 也有违 Nova 的设计架构, 这将致使 Nova 管理范围的扩大, 因此社区尤其 Nova 团队很难接受这种建议。

此外, 社区也有将监控功能集成到 Heat 项目的建议, 但是此方案需要 OpenStack 终端用户在实例故障时使用 Heat 模板来重启实例 (但这应该是云管理员的任务, 而不是要求用户来执行)。从目前社区对 Nova 的发展规划来看, 对基础架构的监控不应该是 OpenStack 的任务, 而应该从基础架构自身来考虑。就当前的 OpenStack 高可用部署环境而言, Pacemaker 结合 Corosync 是使用最多的服务高可用监控工具, 但是由于历史原因, Corosync 对计算节点的支持数目有限, 不过 Redhat 提出的 Pacemaker_remote 解决了这种限制。当然, 除了主流的 Pacemaker 方案, 也有很多用户基于 Nagios 和 Zookeeper 来实现对 Nova 服务进行监控的方案。

(2) 隔离

隔离在高可用集群中是个非常关键的操作, 所谓的集群“脑裂”通常是由不完善的隔离操作引起的。在 OpenStack 集群的 Nova 服务高可用中, 隔离就是将故障计算节点与集群完全隔离, 使其成为孤立节点。在实例高可用环境中, 计算节点可能因各种原因出现故障情况, 在其他健康节点上重启故障计算节点的实例之前, 必须确保此实例确实已经不存在, 否则可能出现一个 OpenStack 集群中同时出现两个相同实例的情况。更糟糕的是, 如果实例部署在共享存储上, 则会出现同一实例镜像运行两个相同实例的情况, 两个实例同写一份数据通常会导致数据损坏, 而且, 这种情况还会导致同一网络中出现两个相同的 IP 地址。因此, 在高可用软件对故障实例进行恢复之前, 必须对监控程序认为有故障的计算节点进行隔离, 否则必然会对实例造成各种破坏和意想不到的情况。Pacemaker 提供了对集群节点的隔离功能, 如果采用其他集群工具, 则需要自己实现隔离功能。

(3) 恢复

监控到计算节点故障之后, 需要对该节点进行隔离, 隔离完成之后, 便可对用户的实例进行恢复。在 Nova 中, 实现实例恢复的功能主要是 Nova 提供的 Evacuate 命令。当 Evacuate 被调用时, 故障计算节点上的实例会被自动撤离, 并在新的节点上恢复实例。为了完成实例的恢复, 实例应该创建在共享存储上, 作为一种替代方案, 也可以将实例创建

在 Cinder 提供的 Volume 上 (SAN BOOT)。不过, 共享存储或者 Volume 并非 Evacuate 执行成功的前提, 如果未采用上述两种方案创建实例, Evacuate 也会在其他节点上恢复实例, 但这是一种有损恢复 (因为 Evacuate 仅是采用与原实例相同的基础镜像在其他节点上重新创建一个相同的实例, 但是原实例中相对基础镜像做过变更的数据却不能恢复, 用户得到的仅是一个与原实例具有相同 UUID 的新实例)。

目前, OpenStack 没有一套完成的监控、隔离和恢复方案, 因此, 用户必须自己实现服务监控和节点隔离, 同时触发故障计算节点上的 Evacuate 操作。如果使用 Pacemaker 集群资源管理器, 则需要在计算节点上实现一个 Evacuate 的资源代理, 从而允许 Pacemaker 触发节点上的 Evacuate 操作。

8.8.2 Nova 实例高可用之 Evacuate/Rebuild

Nova 提供了 Evacuate API 来隔离故障计算节点上的实例 (Evacuate 也是实现计算节点实例高可用的基础), 在监控到计算节点故障并将其隔离之后, 便可触发 Evacuate 操作以对此节点上的实例进行恢复。本质上来说, Evacuate 是对 Rebuild 功能的扩展, 或者说基于实例 HA 的需求而对 Rebuild 功能进行了扩展, 便是现在的 Evacuate, 而 Rebuild 功能仍然具有其实用性。Rebuild 与 Evacuate 的区别主要在于: Rebuild 会刷新虚拟机镜像磁盘, 即使使用新的镜像重新创建具有相同 ID 的实例, 因此 Rebuild 无须共享存储即可实现, 其功能更像是相同的硬件重新安装另外的操作系统 (如 windows 系统换成 Linux 系统); 而 Evacuate 是真正的原样复原, 包括系统和用户数据。此外, Evacuate 和 Rebuild 仅支持 Active 和 Stopped 状态的实例, 而不支持 Paused、Suspend 和 Shutdown 等状态的实例。

Evacuate 的具体操作过程取决于实例的配置和底层存储架构。如果实例是基于计算节点本地文件系统的“临时 (ephemeral)”实例, 则 Evacuate 操作采用与原实例相同的镜像在其他节点创建新的实例, 新实例与原实例具有相同的省份和配置, 如相同的实例 ID、镜像 ID、Flavor、IP、Attached Volumes 等。如果实例位于共享存储上, 则 Evacuate 将在其他节点上基于相同的实例文件重启实例, 并保持实例省份和配置不变。如果实例是基于 Volume 后端的, 则 Evacuate 将重建实例并从相同的 Volume 上启动实例。因此, 基于共享存储和 Volume 后端的实例可以被原样恢复, 而基于本地临时存储的实例不能恢复用户数据 (位于临时存储上的数据)。为了说明和演示 Rebuild 操作与 Evacuate 操作的差异和不同, 下面对 nfs-server 实例分别进行这两个操作, 操作过程中, 实例挂载有 1GB 大小的 Volume。

1. Rebuild 操作

1) Rebuild 前查看实例信息。记录 UUID、IP 地址、主机名和 Volume 挂载情况。

```
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State	Networks
...d4e0	nfs_server	ACTIVE	-	Running	admin-net=192.128.1.19

```

+-----+-----+-----+-----+-----+-----+
[root@controller1 ~]# cinder list
+-----+-----+-----+-----+-----+-----+
| ID | Status | Name | Size | Volume Type | Bootable | Attached to |
+-----+-----+-----+-----+-----+-----+
| ...0894 | in-use | volume1 | 1 | - | false | ...873d4e0 |
+-----+-----+-----+-----+-----+-----+

```

2) 执行 Rebuild 操作。这里仍然使用原镜像 cirros-0.3.4-x86_64 对 nfs-server 进行 Rebuild 操作，用户可以选择任意其他镜像进行 Rebuild 操作。

```

[root@controller1 ~]# nova rebuild nfs_server cirros-0.3.4-x86_64 --poll
+-----+-----+-----+-----+-----+-----+
| Property | Value |
+-----+-----+-----+-----+-----+-----+
| OS-DCF:diskConfig | MANUAL |
| accessIPv4 | |
| accessIPv6 | |
| admin-net network | 192.128.1.19 |
| adminPass | g56CsQdzTTW3 |
| created | 2016-09-13T03:38:14Z |
| description | - |
| flavor | m1.tiny (1) |
| hostId | .....cf6faadcc584b |
| id | 3be393f3-04f6-4437-b0e3-c2cbf873d4e0 |
| image | cirros-0.3.4-x86_64 (.....65df7f942857) |
| locked | False |
| metadata | {} |
| name | nfs_server |
| progress | 0 |
| status | REBUILD |
| tenant_id | cbe811690f3c432aa59fbdcf918a793 |
| updated | 2016-09-13T14:58:06Z |
| user_id | 718d52d937a44d26b981296e71dfbe57 |
+-----+-----+-----+-----+-----+-----+
Server rebuilding... 100% complete
Finished

```

3) Rebuild 之后检查实例信息。注意观察主机名、UUID、IP 地址和 Volume 挂载情况是否改变，正常情况下 Rebuild 后这些参数应该保持不变。

```

[root@controller1 ~]# nova list
+-----+-----+-----+-----+-----+-----+
| ID | Name | Status | Task State | Power State | Networks |
+-----+-----+-----+-----+-----+-----+
| ...d4e0 | nfs_server | ACTIVE | - | Running | admin-net=192.128.1.19 |
+-----+-----+-----+-----+-----+-----+
[root@controller1 ~]# cinder list
+-----+-----+-----+-----+-----+-----+
| ID | Status | Name | Size | Volume Type | Bootable | Attached to |
+-----+-----+-----+-----+-----+-----+

```

```

|...0894 | in-use | volume1 | 1 | - | false |...873d4e0 |
+-----+-----+-----+-----+-----+-----+

```

2. Evacuate 操作

1) Evacuate 之前在实例系统中保存用户数据，以备 Evacuate 后验证。

//在实例系统中保存用户数据到test.txt文件中

```
[root@nfs-server ~]# echo "This data should be recovery after evacuate">\
```

```
/root/test.txt
```

```
[root@nfs-server ~]# more /root/test.txt
```

```
This data should be recovery after evacuate
```

2) Evacuate 前检查实例主机。核实实例的宿主机并确认 Evacuate 操作的目标主机，Evacuate 要求使用共享存储或者基于 Volume 的实例，此处的 nfs-server 为基于 NFS 共享存储的实例。

```
[root@controller1 ~]# nova list
```

```

+-----+-----+-----+-----+-----+-----+
|      ID | Name      | Status | Task State | Power State | Networks      |
+-----+-----+-----+-----+-----+-----+
|...d4e0 | nfs-server| ACTIVE | -          | Running    | admin-net=192.128.1.19|
+-----+-----+-----+-----+-----+-----+

```

```
[root@controller1 ~]# nova hypervisor-servers compute1
```

```

+-----+-----+-----+-----+-----+-----+
|      ID | Name      | Hypervisor ID | Hypervisor Hostname |
+-----+-----+-----+-----+-----+-----+
|...d4e0 | instance-0000001a | 1             | compute1            |
+-----+-----+-----+-----+-----+-----+

```

```
[root@controller1 ~]# nova hypervisor-servers compute2
```

```

+-----+-----+-----+-----+-----+-----+
|      ID | Name      | Hypervisor ID | Hypervisor Hostname |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+

```

3) 计算节点正常运行时执行 Evacuate 操作。Evacuate 操作要求实例宿主机的计算服务不能处于 Active 状态，否则不允许执行。

//compute1节点正常运行，不允许执行Evacuate

```
[root@controller1 ~]# nova evacuate nfs_server compute2
```

```
ERROR (BadRequest): Compute service of compute1 is still in use. (HTTP 400)
```

```
(Request-ID: req-4df60be4-2171-4f91-bc83-84d763069265)
```

4) 计算节点故障时执行 Evacuate 操作。通过 shutdown 计算节点 compute1 来模拟节点故障，注意观察 Evacuate 执行过程中实例状态的变化。

//shutdown compute1节点

```
[root@compute1 ~]# poweroff
```

//执行evacuate，目标节点为compute2

```
[root@controller1 ~]# nova evacuate nfs_server compute2
```

```
[root@controller1 ~]# nova list
```



```
+-----+
+-----+
| ID | Name | Status | Task State | Power State |
+-----+
Networks |
+-----+
```

```
+-----+
|...d4e0 | nfs_server | REBUILD | rebuild_block_device_mapping | NOSTATE |
admin-net=192.128.1.19 |
+-----+
```

```
[root@controller1 ~]# nova list
```

```
+-----+
+-----+
| ID | Name | Status | Task State | Power State |
+-----+
Networks |
+-----+
```

```
+-----+
|...d4e0 | nfs_server | REBUILD | rebuild_spawning | NOSTATE |
admin-net=192.128.1.19 |
+-----+
```

```
[root@controller1 ~]# nova list
```

```
+-----+
+-----+
| ID | Name | Status | Task State | Power State |
+-----+
Networks |
+-----+
```

```
+-----+
|...d4e0 | nfs_server | ACTIVE | - | Running |
admin-net=192.128.1.19 |
+-----+
```

5) Evacuate 后查看实例主机变化情况。实例宿主机应该由 compute1 漂移到 compute2。

```
[root@controller1 ~]# nova list
```

```
+-----+
+-----+
| ID | Name | Status | Task State | Power State |
+-----+
Networks |
+-----+
```

```
+-----+
|...d4e0 | nfs_server | ACTIVE | - | Running |
admin-net=192.128.1.19 |
+-----+
```

```
[root@controller1 ~]# nova hypervisor-servers compute1
```

```
+-----+
+-----+
| ID | Name | Hypervisor ID | Hypervisor Hostname |
+-----+
```

```
[root@controller1 ~]# nova hypervisor-servers compute2
```

```
+-----+-----+-----+-----+
|      ID| Name                | Hypervisor ID | Hypervisor Hostname |
+-----+-----+-----+-----+
|...d4e0 | instance-0000001a | 1              | compute2             |
+-----+-----+-----+-----+
```

6) 验证 Evacuate 后用户数据是否可用。有两个方法可以验证：Evacuate 前更改过实例 root 用户密码，现在检查 root 密码是否仍然可用；Evacuate 前在 /root/test.txt 中保存有用户数据，现在检查数据是否仍然存在。

```
[root@nfs-server ~]# ls -l
```

```
-rw-r--r--  1 root    root      44 Sep 13 23:31 test.txt
```

```
[root@nfs-server ~]# more /root/test.txt
```

```
This data should be recovery after evacuate
```

7) 启动 compute1 节点，检查 compute1 节点上的实例是否还会自动恢复。正常情况下，计算节点上的实例被 Evacuate 操作撤离后，在计算节点恢复过程中会自动清除实例相关信息，被撤离的实例不会在原计算节点上被恢复。

```
[root@controller1 ~]# nova hypervisor-servers compute1
```

```
+-----+-----+-----+-----+
| ID | Name | Hypervisor ID | Hypervisor Hostname |
+-----+-----+-----+-----+
```

```
[root@controller1 ~]# nova hypervisor-servers compute2
```

```
+-----+-----+-----+-----+
|      ID| Name                | Hypervisor ID | Hypervisor Hostname |
+-----+-----+-----+-----+
|...d4e0 | instance-0000001a | 1              | compute2             |
+-----+-----+-----+-----+
```

从上述操作过程可以看到，Evacuate 操作要求实例的宿主机节点计算服务处于非正常运行状态，而且 Evacuate 撤离后的实例以 Rebuild 的方式在目标节点上基于共享存储镜像重新创建，新创建的实例将恢复如初，即实例身份、配置信息和用户数据均不变，同时，当故障计算节点恢复后，其上的实例信息会被自动清除，被 Evacuate 撤离的实例不会恢复。Evacuate 功能的这些特性也是将其用于实例 HA 的关键。

8.8.3 Nova 实例高可用之 Pacemaker_remote

如本章前文和第 2 章中关于不同厂商的高可用方案对比所述，在 OpenStack 的高可用设计方案中，使用最多和部署相对简单的方案是 Pacemaker、Corosync 与 HAproxy 等三方软件结合部署的方案，如 RedHat 和 Maritans 的 OpenStack 发行版本均采用此类集群高可用方案。但是，在采用传统的 Pacemaker 集群方案时，存在 OpenStack 计算节点数目受限的问题，即 Pacemaker 集群中允许的计算节点数目上限为 16 个，即使对于一般规模的 OpenStack 私有云集群环境，这一限制也是很现实的，因此便出现了 Pacemaker_remote

以解决节点受限的问题。

Pacemaker 集群对计算节点的限制并非 Pacemaker 所为，而是集群通信机制 Corosync 对计算节点数目的限制。在基本的 Pacemaker 高可用集群中^①，每个集群节点都运行 Corosync 全集群栈（Full cluster stack）和 Pacemaker 全组件，这种方式虽然为集群带来了灵活性，但是却将集群节点的可扩展性限制在 16 个节点之内。为了允许集群扩展更多节点，Pacemaker 允许未运行全部集群栈的节点集成到集群中，并允许 Pacemaker 像管理正常节点一样管理这些节点上的资源。而这类节点通常称为 Remote 节点或者部分 Pacemaker 节点，因为它们不运行 Corosync 进程，而只运行 Pacemaker_remote 进程，因此不能称为一个完整的 Pacemaker 节点。在运行有 Pacemaker_remote 进程的 Pacemaker 集群中，节点被划分为三种类型，即集群节点，Remote 节点和 Guest 节点。其中，集群节点是运行全部 Pacemaker 组件和 Corosync 集群栈的节点，集群节点可以运行集群资源、运行 Pacemaker 集群命令行工具（crm_mon、crm_resource 等）、执行 Fencing 操作、统计集群 Quorum 和作为集群 DC（Designated Controller）节点。Remote 节点是运行 Pacemaker_remote 进程的物理主机（Pacemaker_remote 是一个允许未运行全栈集群软件节点成为 Pacemaker 管理节点的服务进程），运行 Pacemaker_remote 进程的节点可以运行集群资源和大部分命令行工具，但是不能执行全栈集群节点的其他功能，如执行 Fencing 操作、Quorum 投票和 DC 功能等。Pacemaker_remote 进程是 Pacemaker 本地资源管理进程（Local Resource Management Daemon, LRMD）的增强版本。运行 Pacemaker_remote 的虚拟机称为 Guest 节点。Pacemaker 集群允许虚拟机节点存在，即虚拟机节点资源也可以通过 Pacemaker 集群资源管理器进行管理。Guest 节点与 Remote 节点的主要区别在于 Guest 节点本身就是 Pacemaker 所管理的资源。对于 OpenStack 实例高可用部署，主要用到的是集群节点和 Remote 节点。如 8.7 节所言，Nova 实例高可用环境下对故障计算节点的隔离是十分关键的，在 OpenStack 高可用环境中，Remote 节点通常就是计算节点，而 Pacemaker 的策略引擎（Policy Engine, PE）能够自动处理对 Remote 节点的 Fencing 操作，用户所要确定的是 Remote 节点具有 Fencing 设备。

要使计算节点成为 Remote 节点，只需在计算节点上安装 Pacemaker_remote 软件包并启动 Pacemaker_remote 服务，之后将其加入 Pacemaker 集群即可（基于修复的 Bug 考虑，如果要使用 Pacemaker_remote，建议安装的 Pacemaker 为 1.12 或更高的版本）。Pacemaker_remote 的 yum 安装命令如下：

```
# yum install -y pacemaker-remote resource-agents pcs
```

安装完成之后，创建一个存放 Pacemaker 集群共享授权 Key 的本地目录：

```
# mkdir -p --mode=0750 /etc/pacemaker
# chgrp haclient /etc/pacemaker
```

① 关于 Pacemaker 集群请参考第 3 章。

在 Pacemaker 集群中，所有节点（包括集群节点和 Remote 节点）必须共享同一个授权 Key。如果集群节点已经存在授权 Key，只需将其 Copy 到 Remote 节点即可，如果还没有授权 Key，则通过如下命令创建授权 Key：

```
# dd if=/dev/urandom of=/etc/pacemaker/authkey bs=4096 count=1
```

在 Remote 节点上启动 Pacemaker_remote 服务并设置其为开机自启动：

```
# systemctl enable pacemaker_remote.service
# systemctl start pacemaker_remote.service
```

Pacemaker_remote 启动完成之后，验证是否可以远程连接到 Remote 节点的 3121 端口，通过 SSH 即可连接测试，如果执行 SSH 连接在断开前有如下输出说明连接正常：

```
//假设Remote节点主机名为remotel
# ssh -p 3121 remotel
ssh_exchange_identification: read: Connection reset by peer
```

Remote 节点配置启动完成后，便可将其加入 Pacemaker 集群（这里假设 Pacemaker 集群已经存在）。Remote 节点加入 Pacemaker 集群的过程等同于创建远程连接资源（Remote Connection Resource）。定义远程连接资源的操作需要在 Pacemaker 集群节点上进行（定义远程连接资源本质上属于 Pacemaker 的 crmd 组件），Pacemaker_remote 安装包并没有提供 ocf:pacemaker:remote 资源代理，而是提供了一个可以使用哪些选项的元数据文件 /usr/lib/ocf/resource.d/pacemaker/remote。在 Pacemaker 任一集群节点上执行如下命令，便可分别将多个计算节点（Remote 节点）加入 Pacemaker 集群：

```
//以for循环形式将全部计算节点加入Pacemaker集群
for node in $(echo $compute_node|cut -d " " -f -$nodes_num)
do
    pcs resource create $node ocf:pacemaker:remote\
    reconnect_interval=60 op monitor interval=20
done
```

Remote 节点加入 Pacemaker 集群后，便可在集群中看到集群节点和 Remote 节点：

```
[root@controller2-vm ~]# pcs status
Cluster name: openstack-ha
Last updated: Sat Sep 17 20:13:47 2016      Last change: Sun Jul 3 07:22:15
2016 by hacluster via crmd on controller1-vm
Stack: corosync
Current DC: controller3-vm (version 1.1.13-a14efad) - partition with quorum
5 nodes and 231 resources configured
//此处有三个集群节点和两个Remote节点(compute1和compute2)
Online: [ controller1-vm controller2-vm controller3-vm ]
RemoteOnline: [ computer1 computer2 ]
```

一旦 Remote 节点加入 Pacemaker 集群，则 Remote 节点上的资源就可以像正常集群节点一样被 Pacemaker 管理，包括 Remote 节点资源的创建、启动和停止等。Remote 节点上

的资源创建也在 Pacemaker 集群节点上进行, 而资源是运行在 Remote 节点上还是集群节点上则通过为节点设置资源属性来实现的, 具体操作方式可以参见第3章。这里为 Remote 节点(计算节点)创建如下资源:

```
//运行在计算节点上的neutron-openvswitch-agent资源及约束
# pcs resource create neutron-openvswitch-agent-compute \
systemd:neutron-openvswitch-agent --clone interleave=true --disabled \
--force
# pcs constraint location neutron-openvswitch-agent-compute-clone rule\
resource-discovery=exclusive score=0 osprole eq compute\
# pcs constraint order start neutron-server-api-clone then \
neutron-openvswitch-agent-compute-clone require-all=false
//计算节点libvirt服务及约束
# pcs resource create libvirt-compute systemd:libvirt --clone \
interleave=true --disabled --force\
# pcs constraint location libvirt-compute-clone rule \
resource-discovery=exclusive score=0 osprole eq compute
# pcs constraint order start neutron-openvswitch-agent-compute-clone\
thenlibvirt-compute-clone
# pcs constraint colocation add libvirt-compute-clone with\
neutron-openvswitch-agent-compute-clone
//计算节点openstack-ceilometer-compute服务及约束
# pcs resource create ceilometer-compute \
systemd:openstack-ceilometer-compute --clone interleave=true\
--disabled --force
# pcs constraint location ceilometer-compute-clone rule\
resource-discovery=exclusive score=0 osprole eq compute
# pcs constraint order start ceilometer-notification-clone then\
ceilometer-compute-clone require-all=false
# pcs constraint order start libvirt-compute-clone then \
ceilometer-compute-clone
# pcs constraint colocation add ceilometer-compute-clone with\
libvirt-compute-clone
//nova-compute服务使用的NFS共享存储服务
# pcs resource create nova-compute-fs Filesystem\
device="$master:$nfs_dir/instances" \
directory="/var/lib/nova/instances" fstype="nfs" options="v3" op start\
timeout=240 --clone interleave=true --disabled --force\
# pcs constraint location nova-compute-fs-clone rule\
resource-discovery=exclusive score=0 osprole eq compute
# pcs constraint order start ceilometer-compute-clone then \
nova-compute-fs-clone
# pcs constraint colocation add nova-compute-fs-clone with \
ceilometer-compute-clone
//计算节点nova-compute服务及约束
# pcs resource create nova-compute ocf:openstack:NovaCompute\
auth_url=http://$vip_keystone:35357 username=admin password=admin\
tenant_name=admin op start timeout=300 --clone interleave=true\
--disabled --force
# pcs constraint location nova-compute-clone rule\
```



```
resource-discovery=exclusive score=0 osprole eq compute
```

Remote 节点资源创建完成后，由于对其进行了约束限制，Remote 节点的资源仅会在 Remote 节点启动，而不会在 Pacemaker 集群节点上启动，但是可以在集群节点上对其进行控制管理，就如运行在 Pacemaker 集群节点上的资源一样。Pacemaker 集群资源启动之后，可以看到如下的 Remote 节点资源运行情况：

```
[root@controller2-vm ~]# pcs status
.....
Clone Set: neutron-openvswitch-agent-compute-clone
[neutron-openvswitch-agent-compute]
    Started: [ computer1 computer2 ]
    Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: libvirtd-compute-clone [libvirtd-compute]
    Started: [ computer1 computer2 ]
    Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: ceilometer-compute-clone [ceilometer-compute]
    Started: [ computer1 computer2 ]
    Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-compute-fs-clone [nova-compute-fs]
    Started: [ computer1 computer2 ]
    Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-compute-clone [nova-compute]
    Started: [ computer1 computer2 ]
    Stopped: [ controller1-vm controller2-vm controller3-vm ]
.....
```

可以看到，被约束在计算节点上运行的资源在集群节点 controller1-vm、controller2-vm 和 controller3-vm 上是不会启动的，仅在计算节点 computer1 和 computer2 上启动。通过引入 Pacemaker 和 Pacemaker_remote，计算节点服务运行状况可以被 Pacemaker 监控管理。现在，为了实现对计算节点的远程 Fencing 和虚拟机的 Evacuate 功能，还需创建 Fencing 和 Evacuate 相关的资源。在 Redhat 的 RDO 方案中，Fencing 和 Evacuate 资源的创建如下：

```
pcs resource create nova-evacuate ocf:openstack:NovaEvacuate\
auth_url=http://$vip_keystone:35357 username=admin password=admin\
tenant_name=admin
pcs stonith create fence-nova fence_compute\
auth-url=http://$vip_keystone:35357 login=admin passwd=admin\
tenant-name=admin record-only=1 action=off --force
```

为了可以正常使用 ocf:openstack:NovaEvacuate 和 fence_compute 资源代理，用户需要安装满足如下条件的软件包：

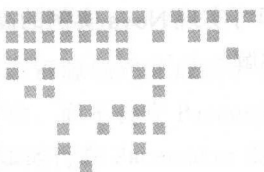
- ❑ fence-agents-4.0.11-13.el7_1.1 或更高版本。
- ❑ resource-agents-3.9.5-40.el7_1.5.x86_64 或更高版本。
- ❑ pacemaker-1.1.12-22.el7_1.4.x86_64 或更高版本。

此外，为了实现 Nova 实例的 HA，实例必须创建在共享存储上。通过上述集群软件安

装和配置，并在 Pacemaker 集群中创建相关的资源，即可实现 Nova 实例的高可用部署。具体的集群部署和实例高可用验证将在后续章节中进行讲解。

8.9 本章小结

Nova 项目是 OpenStack 中最为核心的项目之一，也是 OpenStack 所有项目中成熟度最高和部署使用率最高的项目，同时还是 OpenStack 最初的两大项目之一，因此 Nova 的重要性不言而喻。本章从 OpenStack 项目和社区介绍开始，讲述了 OpenStack 项目的发展历史和未来前景，对 OpenStack 项目各个版本的发行及相关的 OpenStack 峰会，以及 OpenStack 社区组织机构进行了介绍。此外，本章还重点介绍了 Nova 服务项目的设计架构和构成组件，以及与 Nova 相关的各种计算分区和区域划分。云计算的基石是虚拟化，Nova 的主要功能便是对各种 Hypervisors 进行控制管理，为此，本章介绍了常见虚拟化方式和各种 Hypervisor。为了理解 Nova 的主机调度策略，本章对 Nova 的 Scheduler 组件调度原理进行了深入分析，并对 Nova 实例的创建流程和 Nova 实例的各种状态进行了详细介绍。在日常运维中，实例维护和迁移是经常遇见的问题，本章对 Nova 实例的在线和离线迁移做了介绍分析，同时对 Nova 实例的高可用配置部署也进行了深入介绍。



Chapter 9

第9章

OpenStack 网络服务

不论是传统数据中心还是云计算数据中心，网络都是不可或缺的核心资源。Neutron 便是 OpenStack 云计算中的网络服务项目，其源自早期的 Nova-network 网络组件。Nova-network 从 Nova 项目中独立出来之后，社区成立了针对网络功能虚拟化的 Quantum 项目。但由于商标侵权的原因，在 Havana 版本后，Quantum 项目更名为现在的 Neutron 项目。Neutron 是 OpenStack 的核心项目之一，也是 OpenStack 核心项目中成熟相对较晚的项目，但是这完全不妨碍 Neutron 项目在社区的热度，在最近发行的 OpenStack 版本中，Neutron 是新增功能和问题修复最多的核心项目。随着 OpenStack Mtiaka 版本的发行，Neutron 在解决网络高可用性和集群扩展带来的网络性能方面都取得了实质性进展，这为 Neutron 真正实现大规模生产环境高可用集群网络提供了坚实的基础，也是 Neutron 迈向成熟项目的标志。本章将对 Neutron 项目进行架构原理的讲解，并重点阐述实现 Neutron 所支持的各种网络类型以及 Neutron 网络服务的各种高可用方案。

9.1 Neutron 网络概述

OpenStack 的网络服务由 Neutron 项目提供，Neutron 允许 OpenStack 用户创建和管理网络对象，如网络、子网、端口和路由等，而这些网络对象正是其他 OpenStack 服务正常运行所需的网络资源。Neutron 项目中实现的各种插件使得用户可以选择不同的网络设备和软件，并为 OpenStack 的网络架构和部署提供极大的灵活性。此外，Neutron 提供了 API Server 以供用户进行云计算网络的定义和配置，而 Neutron 灵活多样的插件使得用户可以借助各种网络技术来增强自己的云计算网络能力。Neutron 还提供了用以配置、管理各种网络服务的 API，如 L3 转发、NAT、负载均衡、防火墙和 VPN 等高级网络服务。在正式部署

使用 Neutron 网络之前，应先掌握与 Neutron 相关的各种网络概念，这些概念是后续了解和
分析 Neutron 网络功能的基础。本节将对这些概念进行简单描述和解释。

（1）API Server

API Server 是 Neutron 的控制器，用户对网络资源的请求全部先交由 API Server 处
理。API Server 接到用户的请求后，会调用后端插件进行具体的任务实现。Neutron 的 API
Server 不仅实现了对 L2 网络和 IP 地址管理（IPAM，IP Address Managment）的支持，还实
现了对 L3 扩展和网关的支持，其中 L3 主要用于对 L2 网络进行路由，而网关主要用于外
网访问。Neutron 包含了很多网络插件，这些插件不仅实现了各种开源虚拟网络技术，还实
现了对各种商业网络设备和技术的支 持，如路由器、物理交换机、虚拟交换机和软件定义
网络控制器等。随着 OpenStack 生态圈和影响力的扩大，几乎全部的网络设备厂商都宣称
支持 OpenStack，因此 Neutron 的后端网络插件和代理也在不断增加，而 API Server 所提供
的网络功能也在不断丰富。

（2）网络插件与代理

Neutron 提供了灵活多样的网络插件和代理供用户选用，以便用户可以部署适合自身
的云环境网络。插件与代理的主要功能在于实现由 API Server 转发的网络资源请求，如网
络端口的插拔、路由增删、网络和子网创建以及提供 IP 地址等。此外，网络插件和代理
的选取依赖于用户特定云环境下网络设备供应商的选用和所采用的网络实现技术。Neutron
项目默认自带多个虚拟和物理网络设备插件与代理，如思科的虚拟和物理交换机、NEC 的
Openflow 产品、Open vSwitch、Linux bridging 和 VMware NSX 产品等。需要指出的是，
在早期的 OpenStack 网络部署中，不能同时使用多种网络插件，只能多中选一，因此在
OpenStack 网络部署中，最常见的代理和插件组合便是 L3、DHCP 代理和某种具体的网络
插件。L3、DHCP 代理和 Open vSwitch 插件方案是当时用户选用较多的插件代理组合方案。
而为了实现对各种网络插件的支持，并可以在不同的节点上使用不同的插件实现网络功能，
社区开发了 ML2 核心插件用以对各式各样的网络插件进行整合。

（3）Flat 网络

为了创建丰富的网络拓扑，Neutron 提供了两种网络类型，即租户网络（Project
Network）和供应商网络（Provider Network）。在租户网络中，每个租户可以创建多个私有
网络，租户可以自定义私有网络的 IP 地址范围，此外，不同的租户可以同时使用相同的 IP
地址或地址段。与租户网络不同，供应商网络由云管理员创建，并且必须与现有的物理网
络匹配。

为了实现不同租户网络互相隔离，Neutron 提供了几种不同的网络拓扑与隔离技术，
Flat 网络便是其中之一。在 Flat 网络中，不同计算节点上的全部实例接入同一个大二层网
络内，全部实例共享此网络，不存在 VLAN 标记或其他网络隔离技术。接入 Flat 网络的全
部实例通过数据中心核心路由接入 Internet，相对于其他的网络类型，Flat 网络是最简单
的网络类型。在 Neutron 的网络实现中，可以同时部署多个隔离的 Flat 网络，但是每个 Flat

网络都要独占一个物理网卡，这意味着要通过 Flat 网络来实现多租户的隔离，尤其是在公有云环境中，这种方法似乎不太现实。

(4) VLAN 网络

VLAN 网络允许用户使用外部物理网络中的 VLAN ID 创建多个租户或供应商网络。VLAN 网络通过 VLAN ID 进行二层网络的隔离，相同 VLAN ID 内部的实例可以实现自由通信，而不同 VLAN 之间的实例通信则需要经过三层路由的转发。由于 VLAN 网络可以实现灵活多样的网络划分与隔离，故 VLAN 网络是生产环境中使用最为普遍的网络。而在云计算环境中，VLAN 网络主要用于私有云环境中，对于大型公有云数据中心，在租户增加的情况下，VLAN ID 的限制是 VLAN 网络的一大弊端。

(5) GRE 和 VxLAN 网络

GRE 和 VxLAN 是一种网络封装协议，基于这类封装协议可以创建重叠 (overlay) 网络以实现和控制不同计算节点实例之间的通信。GRE 和 VxLAN 的主要区别在于 GRE 网络通过 IP 包进行数据传输，而 VxLAN 通过 UDP 包进行数据传输，GRE 或 VxLAN 数据包在流出节点之前会被打上相应的 GRE 或 VxLAN 网络 ID (Segmentation ID)，而在进入节点后对应的 ID 会被剥离，之后再进入节点内部的虚拟网络进行数据转发。

GRE 和 VxLAN 网络中的数据流要进入外部网络，必须配有路由器，而且要将租户网络与外部网络互连，路由器也是必备的。在 GRE 和 VxLAN 网络中，路由器的主要作用在于通过实例浮动 IP 提供外部网络对实例的直接访问。在很多公有云环境中，GRE 和 VxLAN 网络被广泛使用，因此用户在公有云上创建实例后，通常需要向供应商购买或申请通信运营商 IP 和一个虚拟路由器，并将运营商 IP 作为实例浮动 IP，才能通过 Internet 访问自己的公有云实例。

(6) 端口

端口 (Port) 在 OpenStack 网络中是一种虚拟接口设备，用于模拟物理网络接口。在 Neutron 中，端口是网络设备连接到某个虚拟网络的接入点，如虚拟机的 NIC 只能通过端口接入虚拟网络，端口还描述了与网络相关的配置，如配置到端口上的 MAC 和 IP。Neutron 网络中的端口网络和子网如图 9-1 所示。

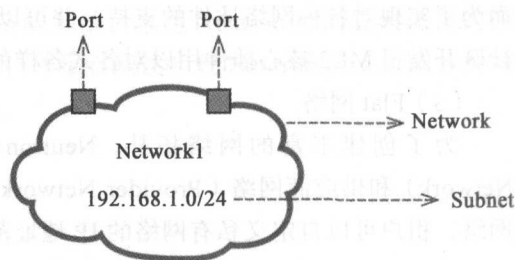


图 9-1 Neutron 中的端口、网络和子网概念

(7) 子网

子网 (Subnet) 代表的是一个 IP 段和相关的配置状态。子网 IP 和网络配置信息通常被认为是网络服务为租户网络和供应商网络提供的原生 IPAM。当某个网络上有新的端口被创建时，网络服务便会使用子网提供的 IP 段为新端口分配 IP。

(8) 子网池

一般来说，终端用户可以在没有任何约束的情况下使用有效的 IP 创建子网，不过有时

需要为 admin 或租户用户预先定义可用 IP 池，并在创建子网时从此 IP 池中自动分配地址。通过子网池，便可要求每个创建的子网必须在预定义的子网池中，从而约束子网所能使用的 IP。此外，子网池的使用还可以避免 IP 被重复使用和不同子网使用重叠的 IP。

(9) 路由

在 Neutron 中，路由 (Router) 是个用以在不同网络中进行数据包转发的逻辑组件，即路由是个虚拟设备，在特定插件支持下，路由还提供了 L3 和 NAT 功能，以使得外部网络 (不一定是 Internet) 与租户私有网络之间实现相互通信。在 Neutron 中，虚拟路由通常位于网络节点上，而租户私有网络要实现与外部 Provider 物理网络 (或 Public 网络) 的通信，则必须经过虚拟路由。Neutron 网络节点虚拟路由连接租户网络和外部网络的示意如图 9-2 所示。路由通常包含内部接口和外部接口，如图 9-2 所示，位于租户网络中的主机 Host_A 和主机 Host_B 通过内部接口接入路由，并通过路由公共网关 (外部接口) 访问外部 Provider 网络和 Public 中的主机 Host_C。但是 Host_C 要访问租户私网中的 Host_A 和 Host_B 主机，则必须通过路由 DNAT 功能才能实现，因此需要为 Host_A 和 Host_B 主机配置 Floating IP。此外，在 Neutron 网络高可用配置中，L3 的高可用配置是整个网络高可用的重点和难点。

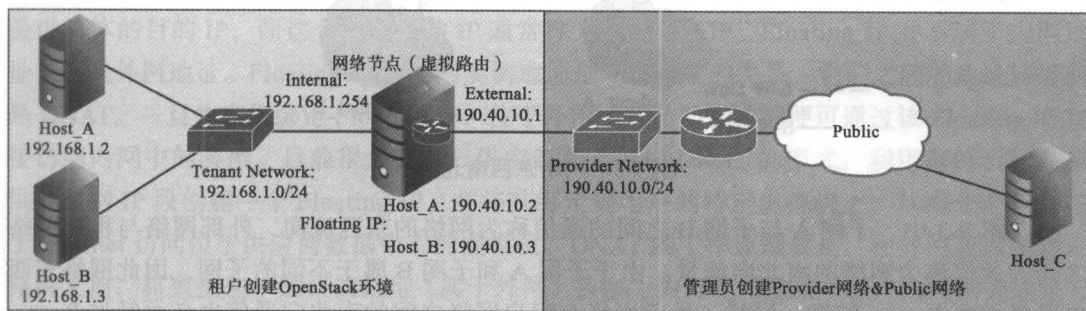


图 9-2 Neutron 虚拟路由

(10) 安全组

安全组 (Security Group) 是虚拟防火墙的规则集合，这些防火墙规则对外部访问实例和实例访问外部的数据包实现了端口级别 (Port Level) 的控制。安全组使用默认拒绝策略 (Default Deny Policy)，其仅包含允许特定数据流通过的规则，每个端口都可以通过附加的形式添加到一个或多个安全组中，防火墙驱动会自动将安全组规则转换为底层的数据包过滤技术，如 iptables。在 Neutron 中，每个项目 (Project) 都包含一个名为 default 的默认安全组，default 安全组允许实例对外的全部访问，但是拒绝全部外网对实例的访问。如果在创建实例时未指定安全组，则 Neutron 会自动使用默认安全组 default。同样，如果创建端口时没有指定安全组，则 default 安全组也会被默认用到此端口。要访问特定实例中某个端口的应用程序，必须在该实例的安全组中开通应用程序要访问的端口。

(11) 网络东西和南北流向

在云计算网络中，一个租户可以有多个租户子网，租户子网通常称为内部网络（简称内网），不同内网中通常会接入不同用途的实例。如用于开发测试环境的实例和用于生产环境的实例通常会分开接入到不同的租户内网中，而租户不同的子网之间因数据访问的需求可能要进行彼此通信，租户内部子网之间的数据访问通常称为东西向通信。此外，位于租户内网中的应用要对外提供服务，则必须实现外部网络与租户网络彼此之间的通信，这包括了外网访问内网和内网访问外网两种方式，通常内外网之间的访问称为南北向通信。网络东西向通信和南北向通信的示意图如图 9-3 所示。

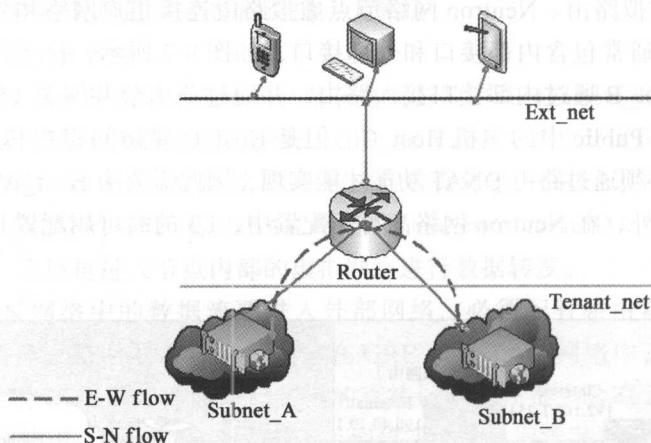


图 9-3 网络东西南北向

在图 9-3 中，子网 A 与子网 B 之间的通信称为网络的東西流向，外部网络与租户网络之间的通信称为网络的南北向流量。由于子网 A 和子网 B 属于不同的子网，因此网络东西向通信需要 Router 转发，同样，外部网络与租户网络之间的南北向通信也必须经过 Router 转发才能实现。在云计算网络环境中，Router 的东西向转发实现了内部子网之间的通信，而南北向转发实现了内部与外部网络之间的通信。

(12) SNAT 源地址转换

源地址转换（Source Network Address Transfer, SNAT）主要用于控制内网对外网的访问，SNAT 通常只需一个外部网关，而无须属于外部网络的浮动 IP（Floating IP），即可实现内网全部实例对外网的访问。在内网存在大量实例的情况下，相对 DNAT，SNAT 可以节约大量外网 IP。在 SNAT 中，尽管内部私网可以访问外网，但是外网却不能访问内部私网，因而 SNAT 具有很好的安全防护机制。很多企业为了防止外网入侵都会使用 SNAT 来实现内网对 Internet 的访问，而在 OpenStack 网络中，SNAT 主要用来实现租户虚拟机实例对外网的访问。SNAT 的工作原理就是，内部私网 TCP/IP 数据包在进入路由后，数据包中的私网源 IP 会被路由上的外网网关 IP 替换，这是源地址转换的核心步骤，即将数据包中源 IP 转换为外网网关地址。这里再次指出，在私有或者公有云中，虚拟机

要访问外网，并不意味着必须为虚拟机分配 Floating IP，而只需创建路由并为路由设置外网网关，将内网接入路由即可实现内网实例对外网的访问。由于 SNAT 是在数据包经过路由之后再进行的 IP 替换，因此 SNAT 又称 POST-Routing，SNAT 的地址转换过程如图 9-4 所示。

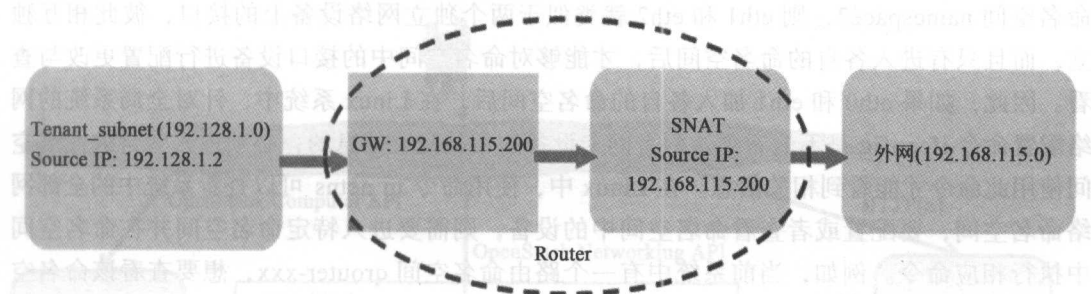


图 9-4 SNAT 地址转换

(13) DNAT 目的地址转换

目的地址转换 (Destination Network Address Transfer, DNAT) 主要用于外网对内网的访问。由于外网要访问位于内网中的某个特定实例，因此必须向位于外网中的访问客户端提供具体的目的 IP，而这个实例目的 IP 通常称为 Floating IP。Floating IP 并不属于内网地址，而是外网地址。Floating IP 与内网实例地址是一一绑定的关系，它们之间的地址转换便是 DNAT。一旦为实例绑定 Floating IP，位于外网中的访问客户端便可通过该 Floating IP 直接访问内网中的实例。目前很多公有云供应商都是通过 DNAT 的形式，利用电信运行商提供的公网 IP 段创建一个 Floating IP，并将其绑定到租户的特定虚拟机上，从而允许租户基于 Internet 访问位于供应商数据中心的虚拟机。DNAT 的工作原理就是，在外网数据包进入路由之前，将数据包中的目的地址 (属于外网) 替换为内部私网地址 (租户内网网关地址)，这也是目的地址转换的核心步骤。经过 DNAT 之后，数据包目的地址被替换并进入路由，路由便将数据包转发到对应的虚拟机。DNAT 过程发生在进入路由之前，即先将外网目的 IP 替换成为内网私有 IP 再进入路由进行转发 (地址替换发生在路由之前)，因此，DNAT 又称 PRE-Routing。DNAT 地址转换过程如图 9-5 所示。

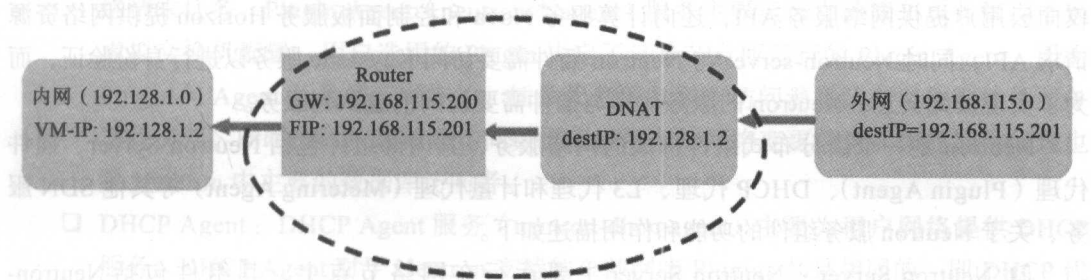


图 9-5 DNAT 地址转换

(14) 网络命名空间

在 Linux 系统中，网络命名空间（Network Namespace）就是一个虚拟的网络设备，网络命名空间有独立的路由表、iptables 策略和接口设备等，网络命名空间彼此之间完全隔离。假设系统中有 eth0 和 eth1 两个网卡设备，且 eth0 属于命名空间 namespace1，而 eth1 属于命名空间 namespace2，则 eth1 和 eth2 就类似于两个独立网络设备上的接口，彼此相互独立，而且只有进入各自的命名空间后，才能够对命名空间中的接口设备进行配置更改与查看。因此，如果 eth0 和 eth1 加入各自的命名空间后，在 Linux 系统中，针对全局系统的网络配置命令 ifconfig 是不能看到这两个网卡设备的相关配置信息的，必须进入各自的命名空间使用此命令才能看到相关信息。在 Linux 中，使用命令 ip netns 可以查看系统中的全部网络命名空间，要配置或者查看命名空间中的设备，则需要进入特定命名空间并在命名空间中执行相应命令。例如，当前系统中有一个路由命名空间 qrouter-xxx，想要查看该命名空间中的接口和 IP 配置情况，可以执行命令：

```
ip netns exec qrouter-xxx ip addr show
```

其中，ip netns exec qrouter-xxx 指明了运行 ip addr show 命令的命名空间。命名空间是 Linux 系统中使用非常广泛的技术，尤其是在网络技术领域，命名空间具有极佳的网络设备模拟能力和配置隔离性。因此在 Neutron 项目中，网络命名空间被大量使用，例如不同的租户网络可以使用重叠的 IP 地址，就是因为不同租户具有独立的路由命名空间。

9.2 Neutron 网络架构

9.2.1 Neutron 网络架构概述

在 OpenStack 的模块架构中，网络是个代号为 Neutron 的独立社区项目，同 Compute、Image 和 Identity 等项目一样，Neutron 的服务部署也需要跨越多个节点。在 OpenStack 中，网络服务使用 Neutron-server 进程提供服务 API 接口，用户通过 Neutron-server 提供的 API 可以进行网络插件的管理配置。通常为了实现网络配置数据的永久性存储，网络插件需要访问 OpenStack 的中心数据库。Neutron 项目的架构如图 9-6 所示，其中 Neutron-server 不仅向云用户提供网络服务 API，还向计算服务 Nova 和控制面板服务 Horizon 提供网络资源请求 API，同时 Neutron-server 与 Neutron 插件需要访问 Keystone 服务以进行身份验证，而为了实现彼此交互，Neutron 的服务代理与插件需要访问消息队列服务。

Neutron 是一个由分布式组件构成的网络服务，其内部组件包括 Neutron Server、插件代理（Plugin Agent）、DHCP 代理、L3 代理和计量代理（Metering Agent）等其他 SDN 服务、关于 Neutron 服务组件的功能和作用描述如下。

- ❑ Neutron Server：Neutron Server 主要运行在网络节点上，其组件包括 Neutron-server 服务进程和 neutron-*-plugin 服务，或者说 Server API 与插件构成了 Neutron

Plugins 进行交互。

- ❑ L3 Agent : L3 Agent 在 Provider 类型的网络中并不是必须的, 而在 Self-Service 网络中却是必须的。L3 Agent 服务主要在外网访问租户网络中的虚拟机时提供 L3 Route 功能。L3 Agent 也需要访问消息队列。
- ❑ Network Provider Service : Network Provider Service 又称 SDN Service, 即网络 SDN 服务, 其主要作用在于向租户网络提供额外的网络服务。SDN Service 通过 REST APIs 通信渠道与 Neutron-server、Neutron-plugin 和 neutron-agent 服务进行交互。

Neutron 内部各个组件代理与 Neutron Server 和 Plugins 之间以 RPC 的形式进行通信, 而 SDN Service 以 REST APIs 的形式访问 Neutron Server 和相应的插件代理。Neutron 内部组件之间的通信数据流和组件逻辑架构如图 9-7 所示。

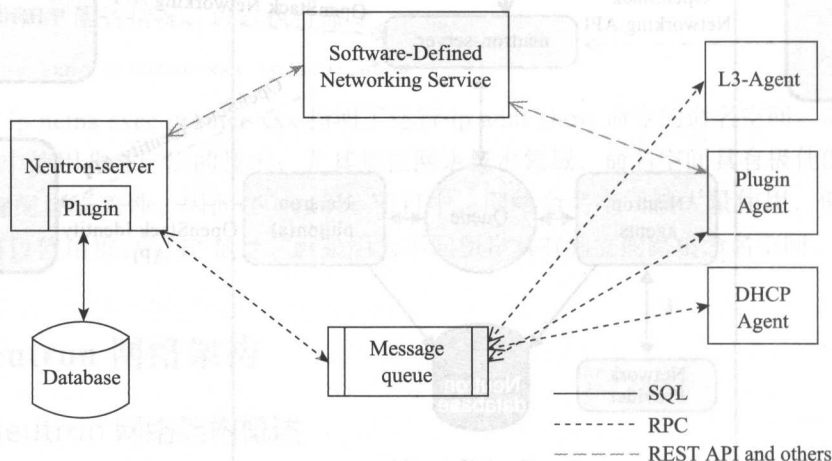


图 9-7 Neutron 内部组件通信架构图

在 Neutron 中, 除了上述介绍的 DHCP Agent 和 L3 Agent, 还有很多提供高级服务的插件代理, 如 VPNaaS 代理、FWaaS 代理和 LBaaS 代理等, 而这些代理通过基于消息队列的 RPC 与 Neutron 的 API Server、Plugins 进行交互。在部署多个 Agent 的 Neutron 网络中, Neutron 服务与 Agent 之间的拓扑关系如图 9-8 所示。

9.2.2 Neutron Plugin 与 Agent

OpenStack 中虚拟机实例所使用的虚拟网络与现实世界中的物理网络非常类似, 为了实现最基本的网络通信功能, 虚拟机实例要求至少存在一个二层网络, 除此之外, 实例可能还要求实现路由、防火墙、VPN 和负载均衡等网络高级功能。而这些网络服务和技术可以通过软件与硬件设备的结合方式来实现, 在具体的技术和产品选型过程中, 又有各种开源软件网络实现技术和不同的厂商设备可供选择。为了可以实现灵活多样的 OpenStack 网络,

Neutron 以 Pulgins 的形式来实现各种网络技术。换句话说, Neutron 之所以能够处理各种网络技术和协议, 主要因为 Neutron 包含了实现各种网络技术的插件。

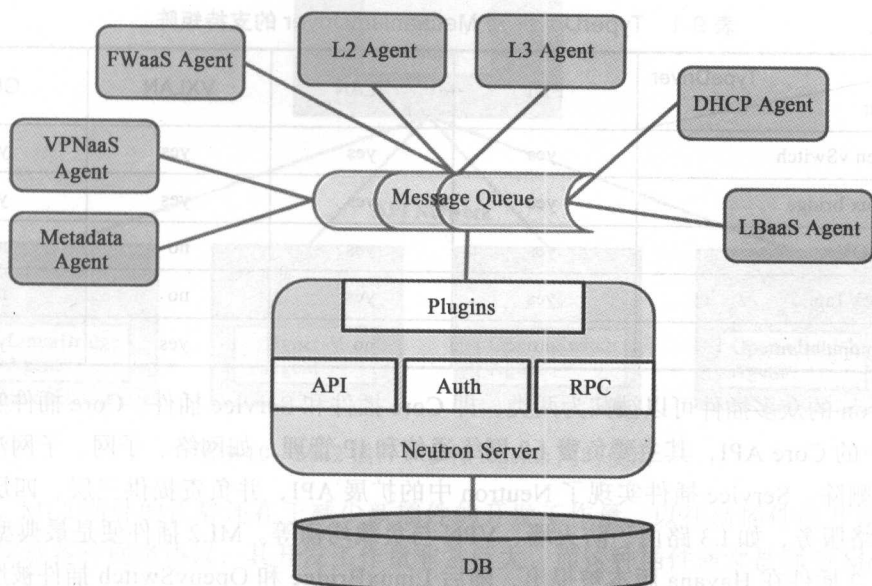


图 9-8 Neutron Server 与 Agent 拓扑

从代码层面而言, 插件是“可插拔 (Pluggable)”的 Python 类和函数, 这些 Python 类在 Neutron API 响应请求时被触发, 同时插件在 Neutron Server 服务启动时加载, 加载完成后插件被当成 Neutron Server 的一部分而运行。在 Neutron 中, API 是 Pluggable 的, 因此用户可以实现自己的 Plugin 并将其“插入”Neutron API 中使用, 通常不同插件实现的 Neutron API 是不同的。用户对插件的实现可以是完整性 (Monolithic) 的也可以是模块式 (Moduler) 的。Monolithic 意味着用户需要实现对网络进行直接或间接控制的全部核心技术, 由于其实现过程较为复杂, Monolithic 形式的插件正被逐步淘汰, 取而代之的是 Moduler 插件, 其中最为成功的便是 Moduler Layer2, 即 ML2 插件。ML2 插件解耦了对不同网络驱动的调用, 它将驱动分为两个部分, 即 TyperDriver 和 MechanismDriver。在 Neutron 网络中, TyperDriver 代表不同的网络隔离类型 (Segmentation Type), 如 Flat、Local、VLAN、GRE 和 VxLAN, TyperDriver 负责维护特定类型网络所需的状态信息、执行 Provider 网络验证和租户网络分配等工作, 而 MechanismDriver 主要负责提取 TyperDriver 所建立的信息并确保将其正确应用到用户启用的特定网络机制 (Networking Mechanism) 中。尽管 ML2 插件是个单一整体的插件, 但是 ML2 插件支持多种 TyperDriver 和 MechanismDriver, 并且不同的 MechanismDriver 所支持的 TyperDriver 种类不一样, 如 LinuxBridge 和 OpevSwitch 两种 MechanismDriver 都支持 Flat、VLAN、VXLAN 和 GRE 这 4 种 TyperDriver, 而 L2 population 仅支持 VXLAN 和 GRE 这 2 种 TyperDriver, 这意味着在启动 L2 population 时, 用户不能使

用 Flat 和 VLAN 进行租户网络的隔离。ML2 插件的 TyperDriver 与 MechanismDriver 彼此支持矩阵如表 9-1 所示。

表 9-1 TyperDriver 对 MechanismDriver 的支持矩阵

MechDriver \ TypeDriver	Flat	VLAN	VXLAN	GRE
Open vSwitch	yes	yes	yes	yes
Linux bridge	yes	yes	yes	yes
SRIOV	yes	yes	no	no
MacVTap	yes	yes	no	no
L2 population	no	no	yes	yes

Neutron 的众多插件可以被归为两类，即 Core 插件和 Service 插件。Core 插件实现的是 Neutron 中的 Core API，其主要负责 L2 网络通信和 IP 管理，如网络、子网、子网池和端口的创建与删除。Service 插件实现了 Neutron 中的扩展 API，并负责提供三层、四层和七层的高级网络服务，如 L3 路由、防火墙、VPN 与负载均衡等。ML2 插件便是最典型的 Core 插件，ML2 插件在 Havana 版本被提出，随后 LinuxBridge 和 OpenvSwitch 插件被废除，并由 ML2 插件来统一取代这些散乱且必须独立使用的完整性插件 (Monolithic Plugins)。自从 H 版本后，Neutron 中已经集成 ML2 插件，并统一使用 ML2 插件来与不同的插件代理协调工作以实现各种 L2 层的网络技术。此外，使用 ML2 插件的最大优势在于，可以并行使用不同开源软件和厂商的网络技术以实现复杂的数据中心网络，而这在 ML2 插件之前是不允许实现的。例如 LinuxBridge 插件和 OpenvSwitch 插件必须独立使用，这种情况如图 9-9 所示。

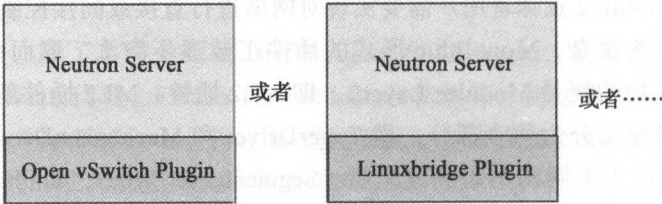


图 9-9 ML2 插件之前的插件独立使用

但是，使用 ML2 插件后，Plugin 必须与 Agent 一一对应的硬性要求被屏蔽，即 ML2 插件可与多个 Agent 协同工作，因此用户便可在不同的节点上使用不同的 Agent 来实现各自的内网机制，如图 9-10 所示。Neutron Server 中启用 ML2 插件，主机 A 使用的是 LinuxBridge Agent，主机 B 使用的是 Hyper-V Agent，而主机 C 和 D 使用的是 Open vSwitch Aagent。

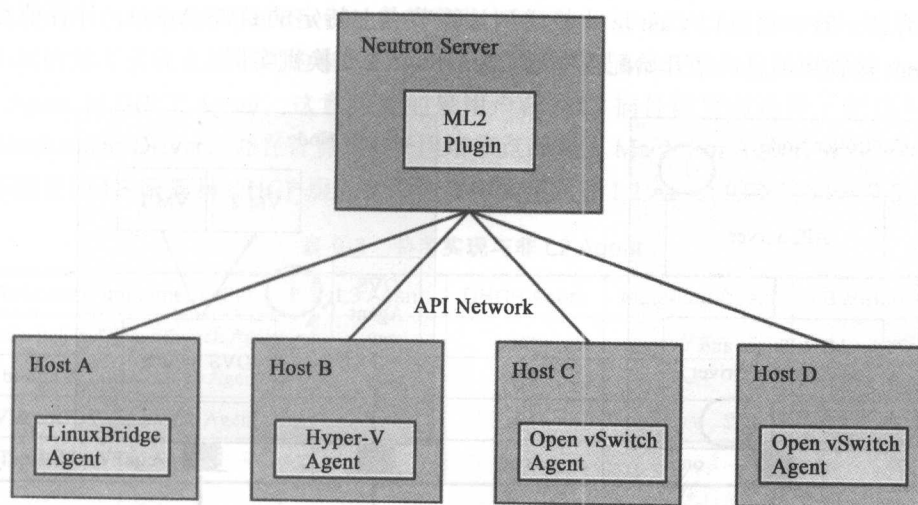


图 9-10 ML2 插件下多种 Agents 同时使用

此外，ML2 插件的优势还在于减少新插件的开发工作量。以往新插件的开发需要从底层开始实现各种网络协议，其开发工作量非常巨大，而这种插件通常被称为 Monolithic Plugins。由于 Monolithic Plugins 需要分开独立开发，因此无法基于已有工作量进行扩展开发，导致开发 Monolithic Plugins 需要做很多重复性的工作而增加额外的工作量。ML2 插件出现后，用户只需要开发相应的 Mechanism Driver 即可，因此工作量大为减少。

在 Neutron 中，Neutron Server 服务扮演的是网络控制中心的角色（通常 Neutron Server 服务也会部署在控制节点上），而实际上与网络相关的命令和配置主要在网络节点和各个计算节点上执行，位于这些节点上的 Agents 便是与网络相关命令的实际执行者，而 Agents 可以划分为 L2 Agents 与非 L2 Agents 两大类，如 OpenvSwitch agent、LinuxBridge Agent、SRIOV nic switch Agent 和 Hyper-V Agent 等属于 L2 Agents。L2 Agents 主要负责处理 OpenStack 中的二层网络通信，是 Neutron 网络中最为核心的部分。非 L2 Agents 主要包括 L3 Agent、DHCP Agent 和 Metadata Agent 等。L3 Agent 负责 L3 层服务，如路由和 Floating IP；DHCP Agent 负责 Neutron 网络的 DHCP 服务，Metadata Agent 允许用户实例通过网络访问 Cloud-init 元数据和用户数据。Agents 所获取的消息或指令来自消息队列总线，并且由 Neutron Server 或 Plugins 发出。由于 Agents 负责网络的具体实现，因此 Agents 与特定的网络技术、Plugins 密切相关，如使用 OpenvSwitch Agent 则意味着通过 OpenvSwitch 技术来实现虚拟网络，并且其对应的插件是 OpenvSwitch Plugin（被 ML2 插件取代）。

图 9-11 所示是 ML2 插件协同 OpenvSwitch Agent 与 Neutron 中的其他网络服务进行交互的简单流程。在图 9-11 所示中，Neutron Server 接收到客户端的 API 请求后，触发 ML2 插件进行请求处理，由于 ML2 插件已经加载了 OVS Mechanism Driver（即 OpenvSwitch 插件），于是 ML2 插件将请求转发给 OVS 驱动，OVS 驱动收到请求后使用其中的可用信息创

建 RPC 消息，RPC 消息以 Cast 形式投递到计算节点上特定的 OVS Agent，计算节点上的 OVS Agent 接收到消息后便开始配置本地 OpenvSwitch 交换机实例。

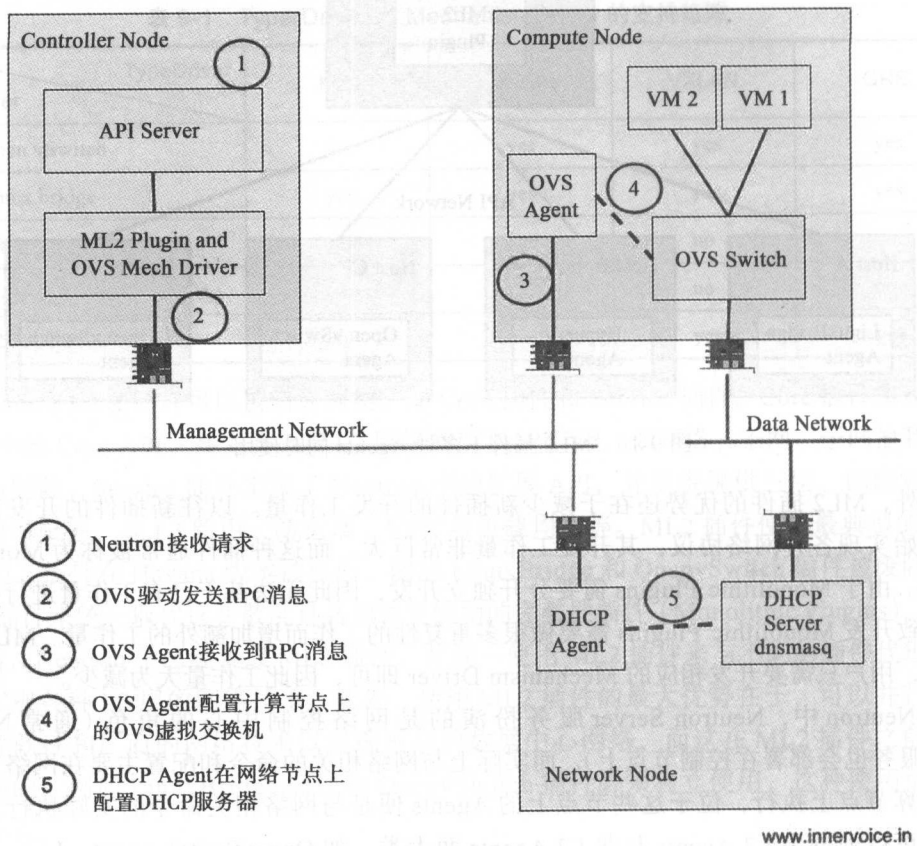


图 9-11 OpenvSwitch Agent 工作流程

在 ML2 插件中，各种 L2 Agent 与 ML2 插件的 Mechanism Driver 有着特定的对应关系，即用户在 ML2 插件配置时选用了特定的 Mechanism Driver 后，相应的计算节点或网络节点上就应该运行特定的 Agent。如在配置 ML2 插件的 Mechanism Driver 参数时，指定了 OVS（代表 OpenvSwitch），则计算节点和网络节点上只能部署和运行 Neutron-OpenvSwitch-agent 服务。表 9-2 给出了不同 Mechanism Driver 与其所对应的 L2 Agent 关系。

通常，将 Mechanism Driver 与 L2 Agent 结合起来称为参考实现，如 Open vSwitch 这个 Mechanism Driver 与 Open vSwitch Agent

表 9-2 Mechanism Drivers 与 L2 Agent

Mechanism Driver	L2 Agent
Open vSwitch	Open vSwitch Agent
Linux bridge	Linux bridge Agent
SRIOV	SRIOV nic switch Agent
MacVTap	MacVTap Agent
L2 population	Open vSwitch Agent, Linux bridge Agent

这个代理的组合被称为 OpenvSwitch 参考实现，用 Open vSwitch & Open vSwitch Agent 表示。不同的参考实现支持不同的非 L2 Agent，如参考实现 MacVTap & MacVTap Agent 不支持 L3 Agent 和 DHCP Agent，这意味着如果用户在 ML2 插件配置时选择了使用 MacVTap 这个 Mechanism Driver，并在计算节点 / 网络节点部署了 MacVTap Agent，则该 Neutron 网络中不能使用 L3 服务和 DHCP 服务。不同参考实现对非 L2 Agent 的支持如表 9-3 所示。

表 9-3 参考实现与非 L2 Agent

Reference Implementation	L3 Agent	DHCP Agent	Metadata Agent	L3 Metering Agent
Open vSwitch & Open vSwitch Agent	yes	yes	yes	yes
Linux bridge & Linux bridge Agent	yes	yes	yes	yes
SRIOV & SRIOV nic switch Agent	no	no	no	no
MacVTap & MacVTap Agent	no	no	no	no

通过上述分析可以看到，Neutron 的功能模块主要由 API Server、Plugin 和 Agent 组成，其中 API Server 和 Plugins 主要由 Neutron-Server 服务运行，并且 Neutron 推荐的核心 Plugin 为 ML2 插件。Neutron 插件的各种 Agent 位于计算节点和网络节点上，负责具体网络技术的实现。如果将 Neutron 的内部功能结构展开，将得到图 9-12 所示的 Neutron 内部结构剖析图。

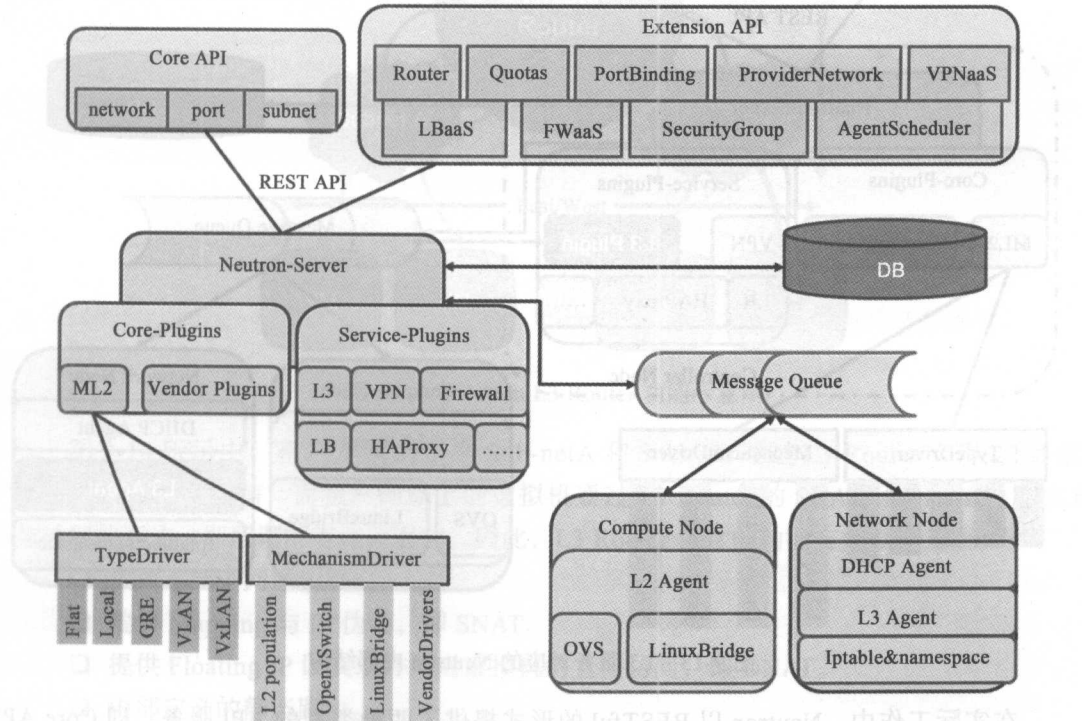


图 9-12 Neutron 内部结构剖析图

9.2.3 Neutron L3 Service 分析

在 Neutron 中, L3 Service 是个非常关键的服务, 其主要提供虚拟机不同 L2 子网之间的 L3 路由, 以及虚拟机与外网之间的 SNAT 和基于 Floating IP 的 DNAT 功能, 如果 OpenStack 网络中未部署 L3 服务, 则用户只能部署基于 Provider 的网络, 而无法实现真正的云计算 Self-Service 网络。L3 Service 主要由 L3 Service 插件及其 API 和对应的 L3 Agent 组成, 其中 L3 Service 插件和 API 由 Neutron-Service 服务来运行, 通常部署在控制节点上, 而 L3 Agent 可以部署在控制节点上, 也可以部署在网络节点上, 但是从 OpenStack 的 Juno 版本开始, L3 Agent 也可以部署在计算节点上以实现分布式虚拟路由 (Distributed Virtual Router, DVR) 功能。或者说, 在 Juno 版本发行之之前, OpenStack 中的 Neutron 网络功能服务除了 Neutron-Server 和 L2 agent 外, 均集中部署在独立的网络节点上。图 9-13 展示了 L3 Service 相关功能模块在 Neutron 内部的部署情况。需要指出的是, Juno 版本之前的 L3 Service 只能按照图 9-13 所示的架构来实现, 即计算节点不能部署 L3 Agent。

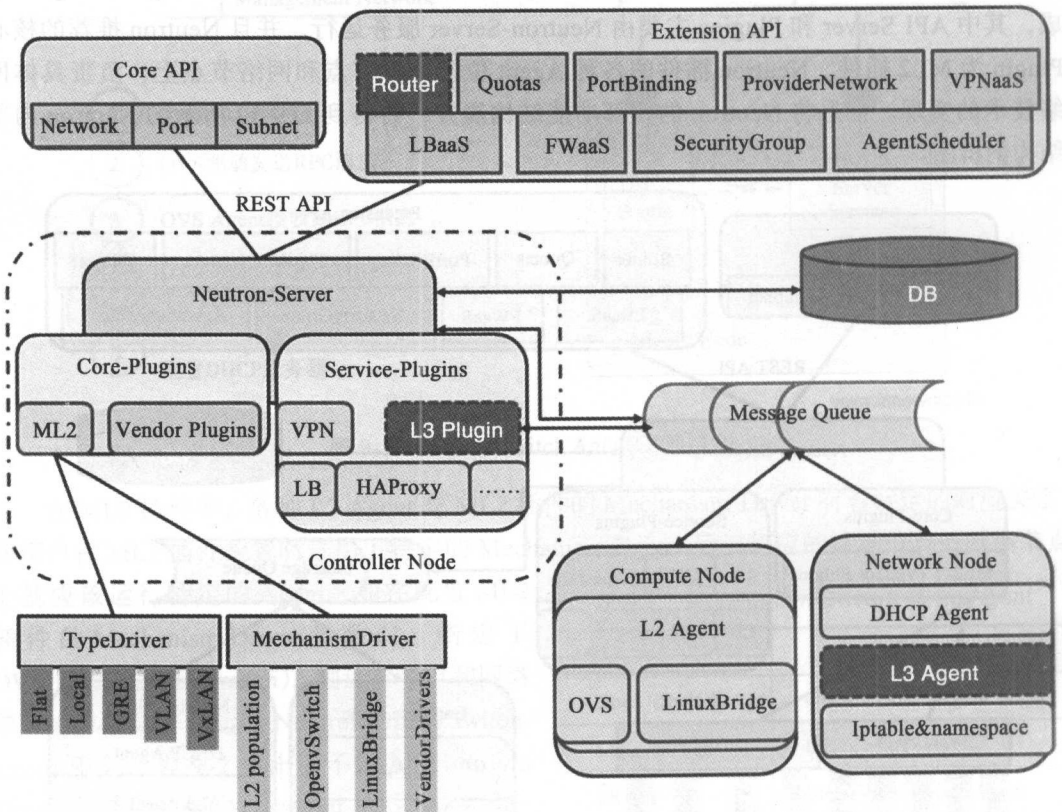


图 9-13 L3 服务对应的 Neutron 内部结构

在实际工作中, Neutron 以 RESTful 的形式提供了两种类型的 API 服务, 即 Core API 和 Extension API。通过 Neutron 的 Core API, 管理员可以创建网络、子网和端口等核心网

络要素。通过 Extension API，管理员可以创建虚拟 Router、LoadBalance、VPN 和 Firewall 等高级的网络功能。与 Neutron 的 Core API 和 Extension API 对应的是 Core Plugins 和 Service Plugins，Neutron Server 中的 API Server 在接收到请求后，会调用相应的 Plugins 来响应用户发起的网络资源请求。Core Plugins 主要用于定义二层网络的网络类型以及网络接口驱动等底层核心功能。Service Plugins 主要用于定义 L3 Router 等三层以上的网络高级功能。通常情况下，租户可以通过 L3 Plugin 提供的 API 创建虚拟 Router 和 Floating IP，并通过 Nova 的 API 将 Floating IP 绑定到实例上，从而实现租户内网与外网的通信。租户网络对外网访问通过 L3 Router 的 SNAT 功能实现，而外网对租户网络的访问通过 L3 Router 的 DNAT 功能实现。除了 NAT 功能，L3 Router 还实现了租户不同子网间的路由功能，Neutron 中 L3 Router 功能如图 9-14 所示。

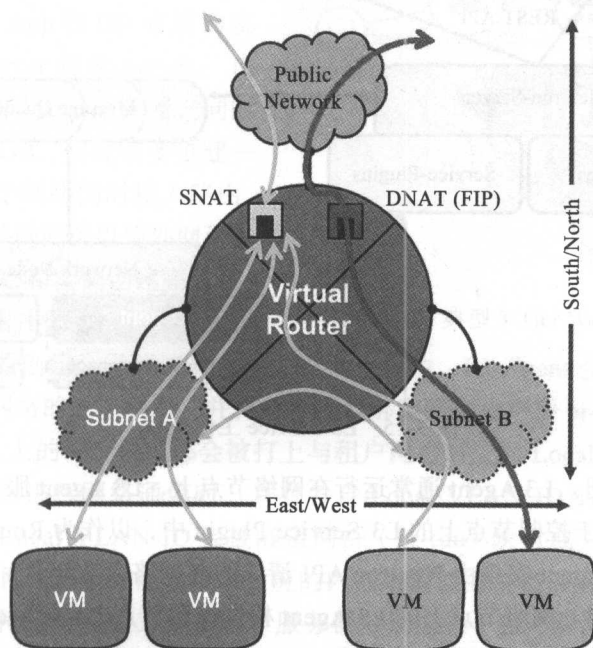


图 9-14 Neutron L3 Router 功能示意图

图 9-14 所示中，租户不同的子网 Sub-netA 和 SubnetB 通过 L3 Router 实现 L2 通信，即所谓的東西向通信；而租户网络上的虚拟机通过 L3 Router 的 SNAT 和 DNAT 功能实现与外网的互访，即所谓的南北向通信。因此，L3 Router 包含的功能有如以下几项：

- ❑ 租户不同子网通信路由。
- ❑ 端口 Mapping 与 IP 伪装，即 SNAT。
- ❑ 提供 Floating IP 以实现外网对虚拟机的直接访问，即 DNAT。
- ❑ 内部定义的静态路由。

结合图 9-13 中所示 L3Service 功能模块的分布和图 9-14 中所示 L3Service 的功能，现

在对 L3 Service 在 Neutron 服务启动过程中的变化进行分析, 并对 L3 Service 的服务过程进行详细梳理。Neutron 中 L3 Service 功能的实现过程大致如下:

1) 插件加载。位于控制节点的 Neutron-Server 在启动时加载 L3 Service 插件, L3 Service 插件主要负责响应和处理用户对三层网络服务的 REST 请求, 并将请求以 RPCcast 形式通过消息队列传递给网络节点上的 L3 Agent 进行最终处理。L3 Service 工作过程如图 9-15 所示。

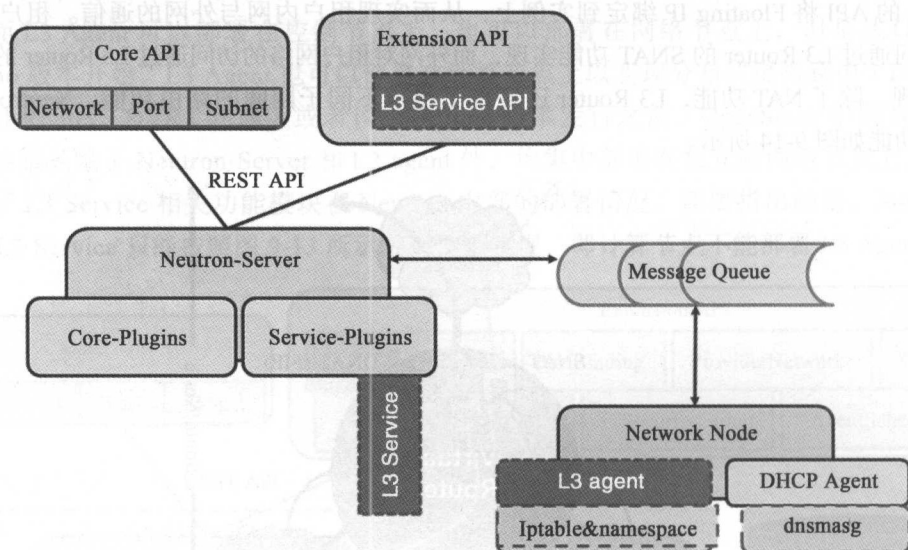


图 9-15 L3 Service 工作流程

2) L3 Agent 注册。L3 Agent 通常运行在网络节点上, L3 Agent 服务启动后, 通过消息队列将自身注册到位于控制节点上的 L3 Service Plugin 中, 以作为 Router Scheduler 对象供 L3 Plugin 调用。L3 Agent 实现由 Neutron API 请求的虚拟路由功能。

3) 虚拟路由管理。网络节点上的 L3 Agent 根据控制节点 L3 Service Plugin 传递的配置参数进行本地 (网络节点) 虚拟路由的管理。

4) 路由功能实现。Neutron L3 Agent 使用 Linux Namespace 和 iptables 规则实现虚拟路由功能。

5) 路由隔离。Namespace 提供了隔离的网络栈 (Network Stack) 空间, 每个网络栈命名空间中都有属于自己的 Routing tables 和 iptables 规则, 由于网络栈命名空间彼此隔离, IP 地址可以在命名空间限制范围内重复使用。

6) 路由创建。L3 Agent 在网络节点上为租户网络中的每一个虚拟路由创建一个命名空间, 之后在命名空间中创建相应的虚拟路由端口。

7) 添加租户子网接口。当用户将租户子网在接入虚拟路由时, 对应命名空间中虚拟路由由接口被标记到集成网桥上 (通常以 br-int 表示), 标记 ID 为租户网络的 Segmentation ID。

8) 设置外网网关。命名空间中的虚拟路由包含一个外网(通常以 br-ex 表示)的网关接口。

9) 路由功能实现。L3 Router 将 iptables 规则应用到命名空间虚拟路由网关接口上以实现 DNAT (Floating IP) 和 SNAT 功能。

为了举例说明 L3 Router 的工作流程, 这里以部署一套常见的三层 (Web-App-DB) Web 应用云虚拟机和租户网络为例。由于要求 Web、App 和 DB 分别处于不同的网络, 因此租户需要分别创建三个网络 web-net、app-net 和 db-net, 同时在三个网络上分别启动用于 Web、App 和 DB 应用的实例 web-server、app-server 和 db-server。由于 Web 服务器需要访问 App 服务器, 同时 App 服务器需要访问 DB, 因此需要创建一个虚拟路由, 并将三个网络同时接入路由, 同时需要为路由设置外网网关以便外网可以访问 Web 服务器。网络和实例创建完成后, 对应的 Neutron 网络拓扑如图 9-16 所示。

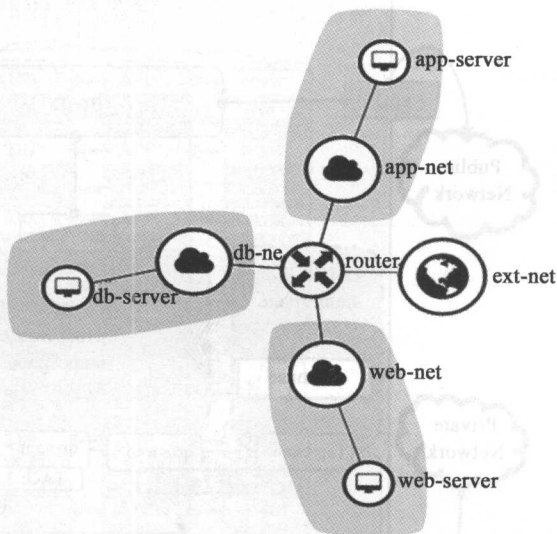


图 9-16 典型 3-Tier Web 应用网络拓扑

在上述虚拟路由的创建、子网接入和网关设置过程中, L3 Agent 会为命名空间中的虚拟路由创建对应三个网络的三个端口, 并将三个端口绑定到集成网桥 br-int 上, 为了进行网络隔离, 绑定到 br-int 上的每个接口都会被打上与租户网络对应的 Local VLAN ID。当各个租户网络跨越物理节点进行网络通信时, 具有各自 Local VLAN ID 的网络数据在进入隧道 (Tunnels) 之前, Local VLAN ID 会被隧道网桥 (br-tun) 映射成为 Segmentation ID。与图 9-16 中所示租户网络拓扑的网络节点对应的内部结构如图 9-17 所示。

图 9-17 所示中, 网络节点中三个 DHCP 服务器分别为三个租户网络的实例提供 DHCP 服务, 而三个租户网络均接入同一个虚拟路由, 只是在 br-int 上创建了三个具有不同 Tag 的端口, 如 qr-<web> (<web> 代表租户 Web 网络 ID)、qr-<app> 和 qr-<db>, 而对应的 VLAN Tag 分别为 1、2 和 3。如果外部网络需要访问租户内网, 则必须经过虚拟路由的网关接口才能实现, 而如果租户内部私网在不同节点之间通信, 则需要经过隧道网桥 br-tun, 并通过 VxLAN 封装协议来实现。在多数情况下, Neutron 的上述 L3 工作方式可以正常应对 OpenStack 灵活复杂的网络要求, 但是这种 L3 功能集中部署的方式存在固有限制, 而这种限制极大影响了整个 OpenStack 集群的性能和可扩展性。在上述 3-Tier Web 应用服务器部署示例中, 除了租户私网与外网的南北向通信外, 租户不同子网之间的通信数据流也要全部经过网络节点, 这就意味着位于不同计算节点上的实例之间进行通信, 其数据流也要进出网络节点, 即所有计算节点的数据流都要先汇聚到网络节点才能进行彼此通信。而正常情况

下, Web 服务器对 App 服务器的网络访问和 App 对 DB 的访问都不应该绕到网络节点, 而是在两台服务器的宿主计算节点之间便可直接进行, 然而由于计算节点没有 L3 Agent, 因此全部计算节点的数据流都必须集中到部署了 L3 Agent 的网络节点, 这个过程如图 9-18 所示。

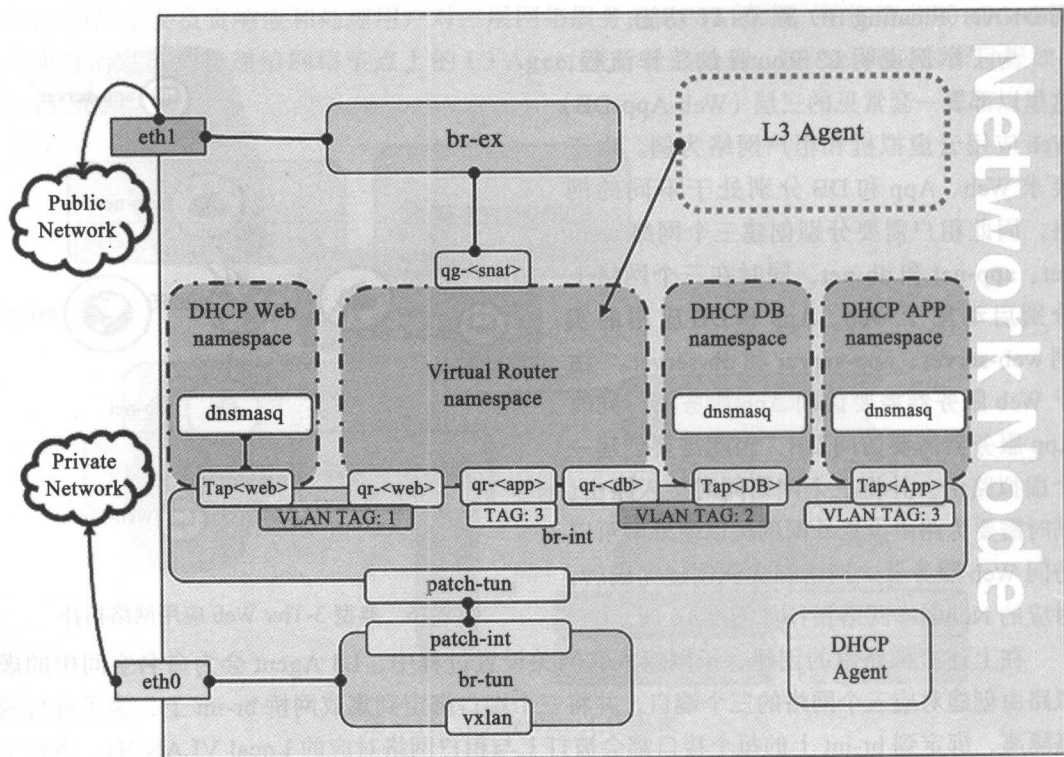


图 9-17 3-Tier Web 应用架构下网络节点内部实现

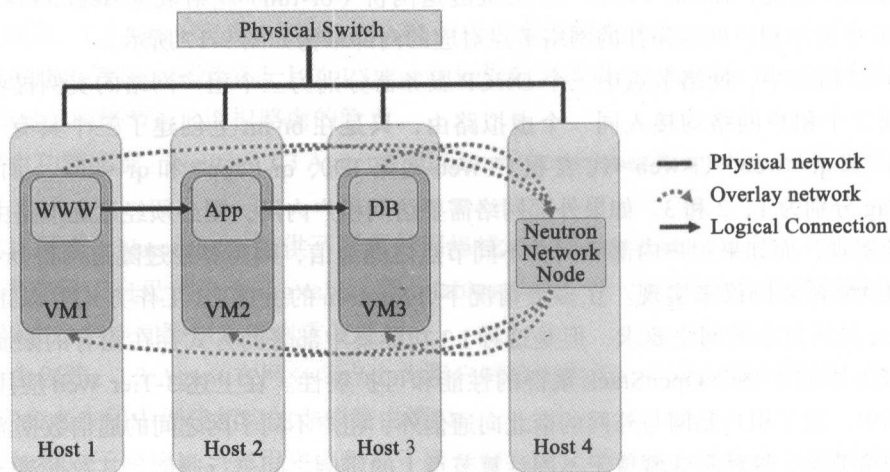


图 9-18 3-Tier Web 应用中计算节点网络汇聚

在小规模的 OpenStack 集群中, 计算节点数目较小的情况下, 上述 L3 Agent 集中的网络部署形式对集群性能的影响可能不是很明显, 但是随着集群规模的扩大, 计算节点数目变得越来越多, 计算节点上的虚拟机之间的通信也变得越来越频繁, 此时的网络负载便会出现质的提升, 因此网络瓶颈将会十分突出, 如图 9-19 所示。

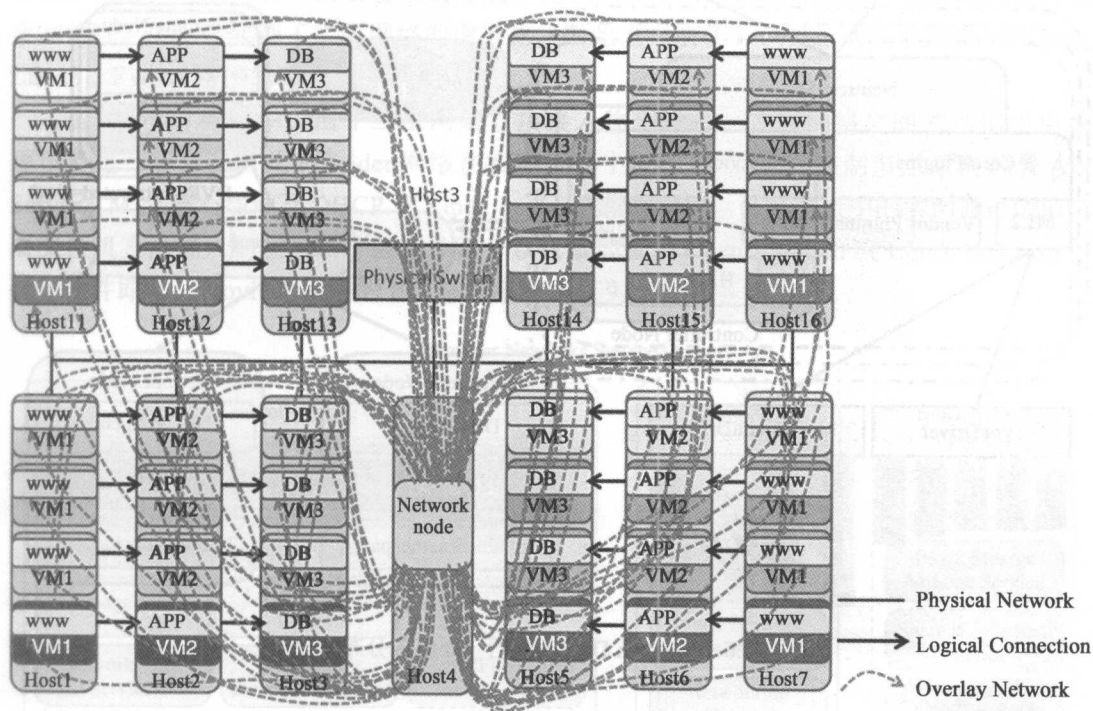


图 9-19 L3 Agent 集中部署下的网络瓶颈

在 L3 Agent 集中部署到网络节点的情况下, 单一的网络节点很可能因为过大的网络负载而出现各种性能问题。为了解决这一问题, 从 Juno 版本开始, 社区引入了分布式虚拟路由 DVR (Distributed Virtual Router) 功能。DVR 的主旨思想就是将 L3 Agent 的功能由网络节点分布到计算节点, 从而让计算节点分流单一网络节点上的网络负载。OpenStack Juno 版本发行后, DVR 功能被实现, L3 Service 的 L3 Agent 可以部署到计算节点, 计算节点实例无须再到网络节点获取 L3 Routing 服务, 网络节点仅负责 L3 SNAT 功能, 其余 L3 Routing 功能被分散到各个计算节点。但是在启用 DVR 模式的情况下, 用户不能启用 Neutron 的 L3 HA 功能。不过在 Mitaka 版本发行后, DVR 与 L3 HA 的冲突限制被解决, 当计算节点启用 DVR 模式时, 网络节点的 L3 SNAT 可以启用基于 VRRP 协议的 L3 SNAT 高可用功能, 即 Mitaka 版本中的 Neutron 实现了 L3 Service 全部功能的高可用。DVR 模式下 L3 Service 各个功能模块在 Neutron 内部的分布情况如图 9-20 所示。

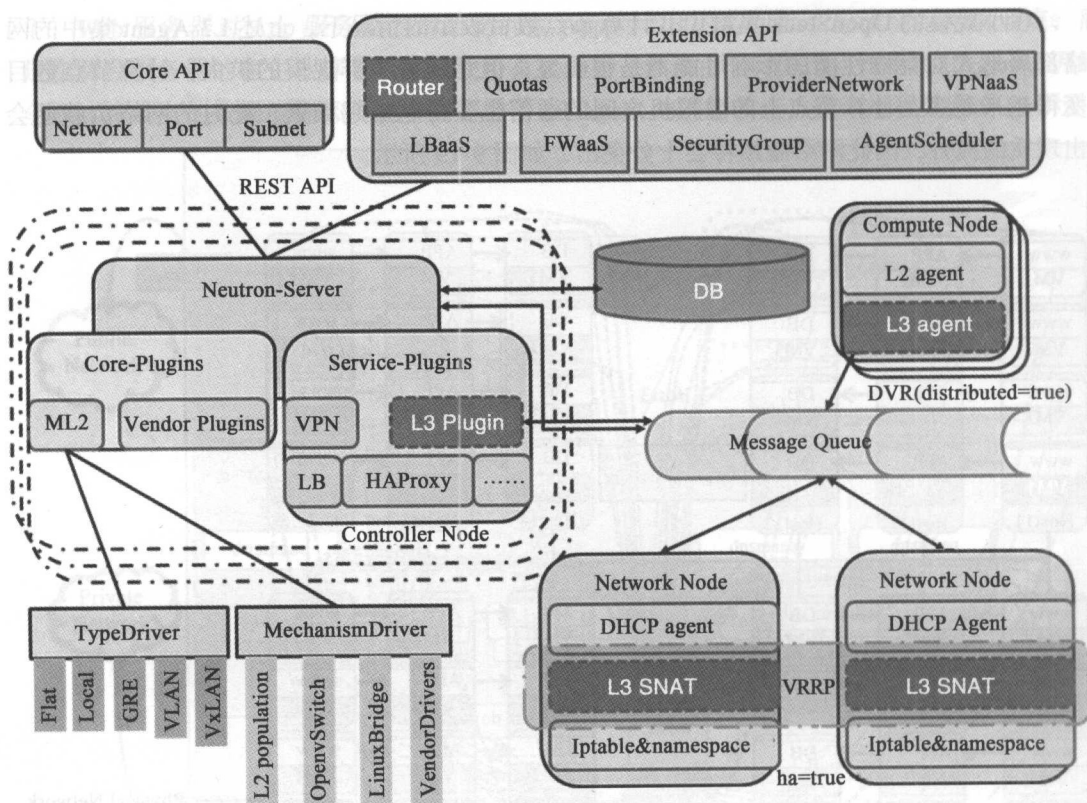


图 9-20 Neutron DVR 模式下 L3 服务分布情况

9.3 Neutron 网络类型

在 Neutron 网络的具体实施过程中，用户可以选择两种网络类型，即 Provider 网络和 Self-Service 网络，后者又称 Tenant 网络或 Project 网络。二者主要区别在于 Provider 网络只能由云管理员创建，并且实例仅可接入 Provider 网络（又称外部网络或物理网络），Provider 网络中不存在租户私网，也没有 L3 路由和 Floating IP 地址，同时 Provider 网络仅支持 Flat 和 VLAN 网络类型；Self-Service 网络是对 Provider 的扩展，在租户可以创建 Self-Service 网络之前必须先由云管理员创建 Provider 网络，Self-Service 网络具有 L3 服务，因此实例可以接入租户私网，除了云管理员，其他非授权用户也可以自助管理和使用 Self-Service 网络，包括用以将私网与外网进行互联的 Router。在 Self-Service 网络中，外部网络（如 Internet）对接入私网的实例访问通过 Floating IP 来实现。

9.3.1 Provider 网络

Provider 网络是一种仅实现二层网络虚拟化，不提供三层路由和更高层的 VPN、

LoadBlancer、Firewall 等高级功能的 Neutron 网络类型,相对而言,Provider 是一种半虚拟化的网络。在 Provider 中,三层以上的功能不被虚拟化,而是借助物理网络设备来实现。Provider 网络部署相对简单,由于使用了物理网络设备来实现高层的网络功能,因此在网络性能和稳定性上可以得到很好的保障,但是 Provider 网络牺牲了 Neutron 网络的灵活性,而且只有经过授权的云管理员才能创建 Provider 网络,而普通租户无法对其进行创建和管理。由于仅实现了二层网络的虚拟化,因此也无法在 Provider 网络中提供 VPNaaS、LBaaS、FWaaS 等网络高级服务。

在 Provider 网络中,由于二层网络直接接入物理设备,二层网络之间的通信也由物理设备进行转发,因此 Provider 网络在实现过程中无须 L3 服务,控制节点只需部署 API Server、ML2 核心插件及 DHCP 和 Linux bridging 代理即可。计算节点中的实例通过虚拟二层交换机直接接入物理网络,因此计算节点只需部署如 Open vSwitch 或 Linux bridging 等代理软件即可,Provider 网络的节点服务布局如图 9-21 所示。

Provider Networks Service Layout

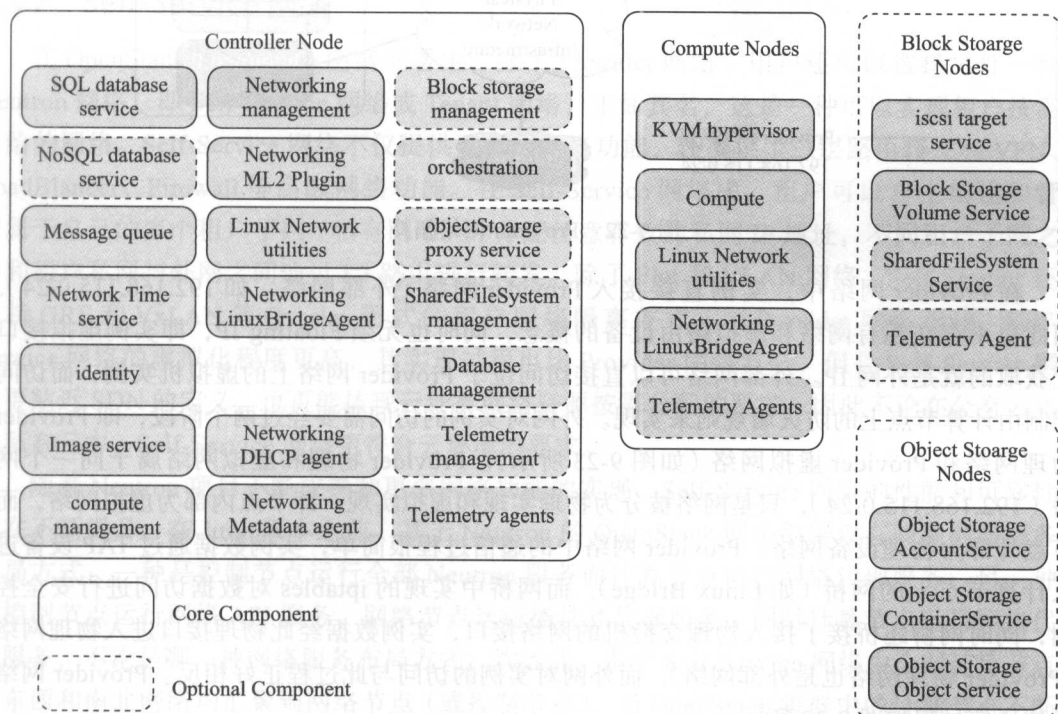


图 9-21 Provider 网络服务布局

Provider 网络拓扑架构很简单,只需在控制节点和计算节点中分别规划一块物理网卡,并将其接入物理网络即可。如果采用的是 VLAN 网络,则将交换机接口配置为 Trunk 模式,节点只需一块物理网卡便可通过多个 VLAN ID 来实现不同网络的隔离。如果采用的是 Flat

网络，由于 Flat 网络没有 Tagging，如要配置多个 Flat 网络则需要节点提供同样数量的物理网卡。Provider 网络的拓扑架构如图 9-22 所示。

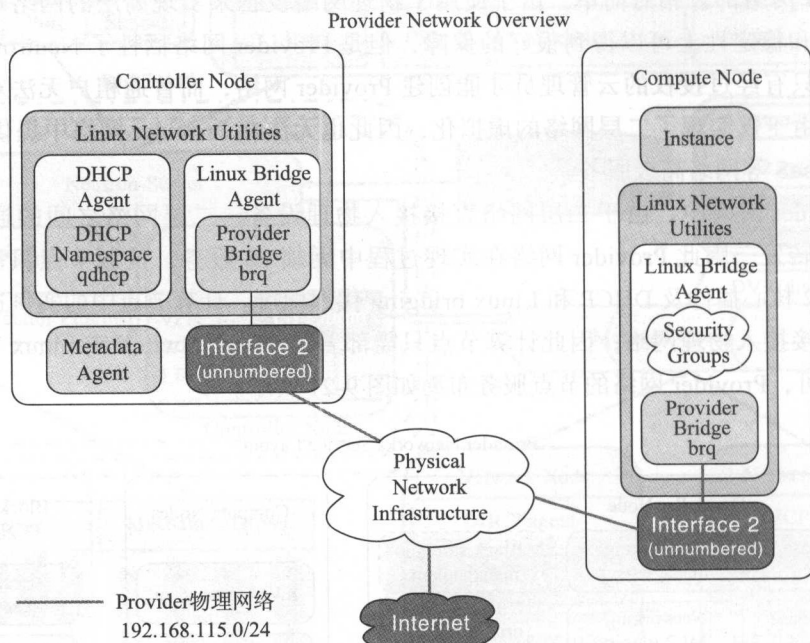


图 9-22 Provider 网络拓扑

在 Provider 网络中，实例直接接入 Provider 网络（外部网络，如 192.168.115.0/24），因此也不存在私有网络和虚拟路由设备的概念，同时也无须 Floating IP，即实例虚拟接口上获取的就是外网 IP。外部网络可以直接访问位于 Provider 网络上的虚拟机实例，而访问控制由计算节点上的防火墙规则来实现。外网对实例的访问需要经过两个阶段，即 Provider 物理网络和 Provider 虚拟网络（如图 9-23 所示）。Provider 物理和虚拟网络属于同一个网络（192.168.115.0/24），只是网络被分为物理实现和虚拟实现，即节点内部为虚拟网络，而节点外部为物理设备网络。Provider 网络中的通信过程很简单，实例数据通过 TAP 设备进入计算节点上的网桥（如 Linux Bridge），而网桥中实现的 iptables 对数据访问进行安全控制，同时网桥还桥接了接入物理交换机的网络接口，实例数据经此物理接口进入物理网络（Provider 物理网络也是外部网络），而外网对实例的访问与此过程正好相反。Provider 网络具体通信过程如图 9-23 所示。

相对 Self-Service 网络，Provider 网络的拓扑架构和通信过程都很简单，因此 Provider 网络在故障排查中也相对容易，并且 Provider 网络中节点之间和二层网络之间的通信都由物理设备负责，因此 Provider 的稳定性和性能比起全虚拟化实现的 Self-Service 网络来说要高很多，此外，Provider 网络也更容易被理解和实现。

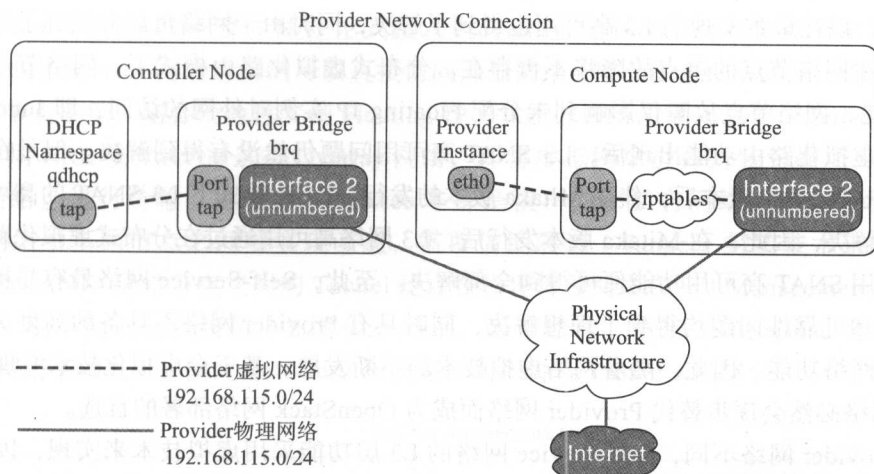


图 9-23 Provider 网络通信

9.3.2 Self-Service 网络

在 OpenStack 的 Neutron 网络部署中,除了 Provider 网络,用户还可以选择另外一种 Neutron 网络,即 Self-Service 网络或 Tenant 网络,正如其名,这是一种可以实现租户按需自给的网络。Self-Service 网络不仅提供了二层网络功能,还提供了三层路由转发和 VPN、LoadBlancer、Firewall 等高级网络功能。在 Self-Service 网络中,租户可以自主创建和管理属于自己的多个租户子网(私有网络)并分配任意有效的私网 IP 地址,不同租户子网之间和租户私网与外网之间通过 L3 路由进行转发。除了 Flat 和 VLAN 网络,Self-Service 还支持 GRE 和 VxLAN 等 Overlay 形式的租户网络隔离方式。同 Provider 网络相比,Self-Service 网络的虚拟化程度更高,其实现过程也比 Provider 网络复杂,但是 Self-Service 网络更贴近 SDN 的定义,也更能体现云计算网络资源按需分配的要求,因此不论在公有云还是私有云中,Self-Service 网络更符合云环境的要求。

随着 Neutron 项目不断成熟和更多网络功能的实现,Self-Service 网络的性能和可靠性也在不断提升。在 Juno 版本之前,基于 Neutron 的 OpenStack 网络部署仅有两种网络服务布局方式:一种是控制节点运行全部 Neutron 服务而计算节点运行网络代理服务;另一种是控制节点运行网络 API 服务。网络节点运行插件和代理服务,同时计算节点运行网络代理服务,不论是哪一种网络服务布局方式,均如此。由于 Self-Service 网络中全部计算节点的东西和南北网络均汇聚到网络节点(或控制节点),当 OpenStack 集群中的计算节点不断增加时,Self-Service 网络的 L3 服务需要承载全部的网络负载,因而很容易导致网络瓶颈出现,从而使整个 OpenStack 集群的性能受到影响。从 Juno 版本开始,Neutron 实现了分布式虚拟化路由(Distribute Virtual Router, DVR)功能,分布式虚拟化路由将网络流向分散到每个计算节点,网络节点上只保留 L3 层的 SNAT 功能。自从 Juno 版本推出分布式虚拟化路由功能后,Neutron 网络节点的瓶颈问题得到了根本性的解决。由于实现了路由的分

布式部署，以往最难实现的 L3 高可用也得到了解决，同时由于网络负载分散到了每个计算节点，以往网络节点的单点故障将不再存在。分布式虚拟化路由模式下，网络节点只剩下 SNAT 功能，网络节点故障仅影响到未分配 Floating IP 实例对外网的访问，即 Juno 版本中的分布式虚拟化路由功能出现后，L3 SNAT 高可用问题仍然没有得到解决。但是在经历了 Kilo 和 Liberty 两个版本后，伴随 Mitaka 版本的发行，DVR 模式下 L3 SNAT 的高可用问题也得到了解决，因此，在 Mitaka 版本发行后，L3 网络高可用通过在分布式虚拟化路由模式下同时启用 SNAT 高可用功能便可得到全部解决。至此，Self-Service 网络最容易被诟病的性能瓶颈和可靠性问题均得到了理想解决，同时具有 Provider 网络不具备的高度灵活性和多种高级网络功能。因此，随着网络虚拟技术的不断发展，基于全虚拟化技术实现的 Self-Service 网络必然会逐步替代 Provider 网络而成为 OpenStack 网络部署的首选。

与 Provider 网络不同，Self-Service 网络的 L3 层功能采用虚拟技术来实现，因此 Self-Service 网络在实现过程中需要运行更多的网络服务组件。由于部署 Self-Service 网络之前，必须首先实现 Provider 网络，因此 Self-Service 网络的服务布局相对于 Provider 主要是多了 L3 Agent，如图 9-24 所示。

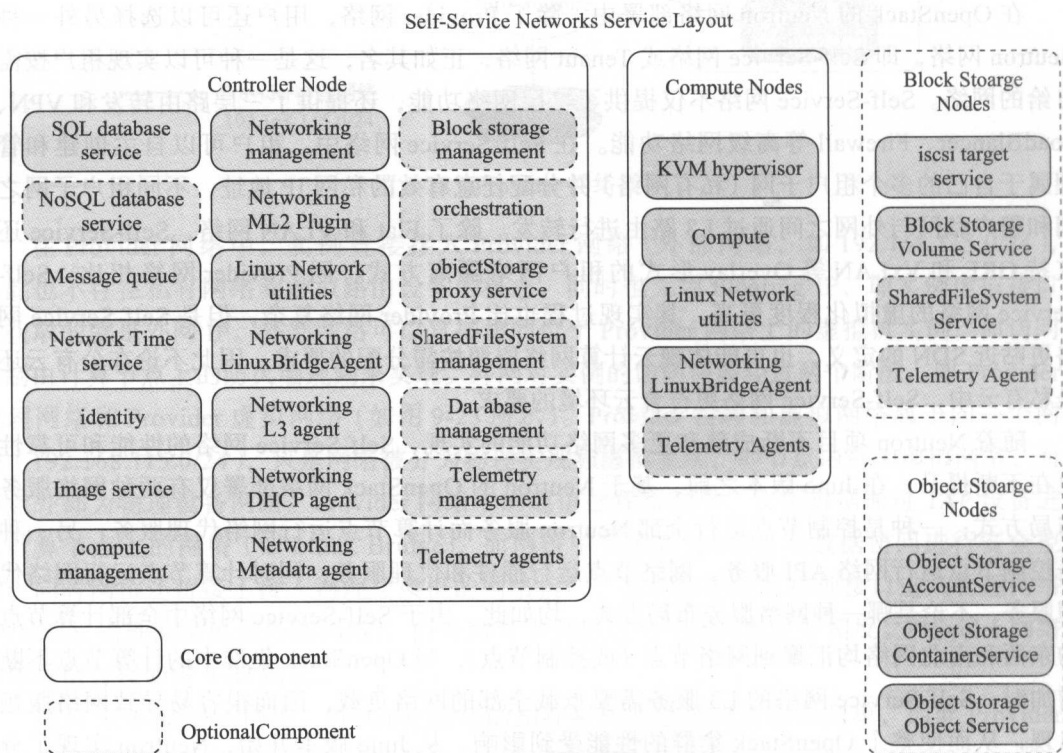


图 9-24 Self-Service 网络服务布局

在租户可以创建 Self-Service 网络之前，管理员必须事先创建 Provider 网络，然后租

户创建的 Self-Service 网络通过 NAT 形式接入 Provider 物理网络。计算节点上的实例创建时，接入 Self-Service 网络的实例可以直接访问外部网络（如 Internet）。但是，外部网络要访问计算节点实例，则必须通过实例 Floating IP，并通过 DNAT 才可实现。Self-Service 网络通常使用 GRE 或 VxLAN 这类 Overlay 网络，即将虚拟网络进行封装后叠加（Overlay）到物理网络上进行传输。由于 Self-Service 本身是租户内部私有网络（虚拟网络），Self-Service 网络中不同节点之间要实现互通，需要借助隧道（Tunnel）封装技术来实现（如 GRE 或 VxLAN），通常节点之间的 Tunnel 建立在物理管理网络之上（Management Physical Network），Self-Service 网络拓扑如图 9-25 所示。图 9-25 中，由于 Self-Service 网络建立在 Provider 网络基础之上，因此 Self-Service 网络中可以并存 Provider 网络。

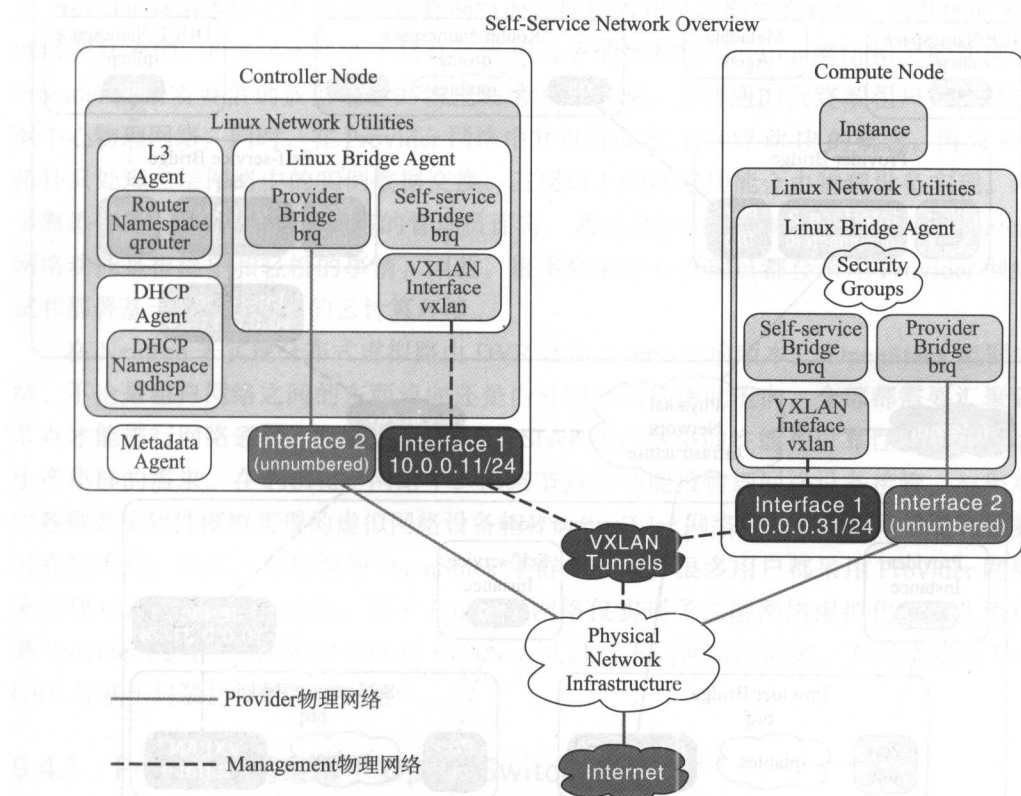


图 9-25 Self-Service 网络拓扑

如前文所述，Provider 网络仅支持 Flat 和 VLAN 网络，而 Self-Service 网络中可以并存 Provider 网络，因此 Self-Service 网络支持 Flat、VLAN、GRE 和 VxLAN 网络类型，并且 Self-Service 网络中的实例也可以直接接入 Provider 物理网络，并使用 Flat 或 VLAN 方式进行网络隔离。一般情况下，在创建 Self-Service 网络后，计算节点实例通常接入 Self-Service 网络，Self-Service 网络再通过 L3 路由接入 Provider 物理网络，并且 Self-Service

中的网络类型通常采用 GRE 或 VXLAN。当实例创建完成并接入 Self-Service 网络后，通常需要在 Provider 物理网络中创建 Floating IP 并将其绑定到实例上，以使得外部网络可以直接访问实例，这里实例从 Self-Service 网络中获取的 IP 称为固定 IP（Fixed ID），而租户从 Provider 物理网络中获取并绑定到实例的 IP 称为浮动 IP。浮动 IP 可以解绑定之后重新分配给其他实例（如果将实例直接接入 Provider 网络，则不存在固定 IP 与浮动 IP 的概念）。Self-Service 网络内部的通信过程如图 9-26 所示。图中不仅有 Self-Service 网络，还存在 Provider 网络。

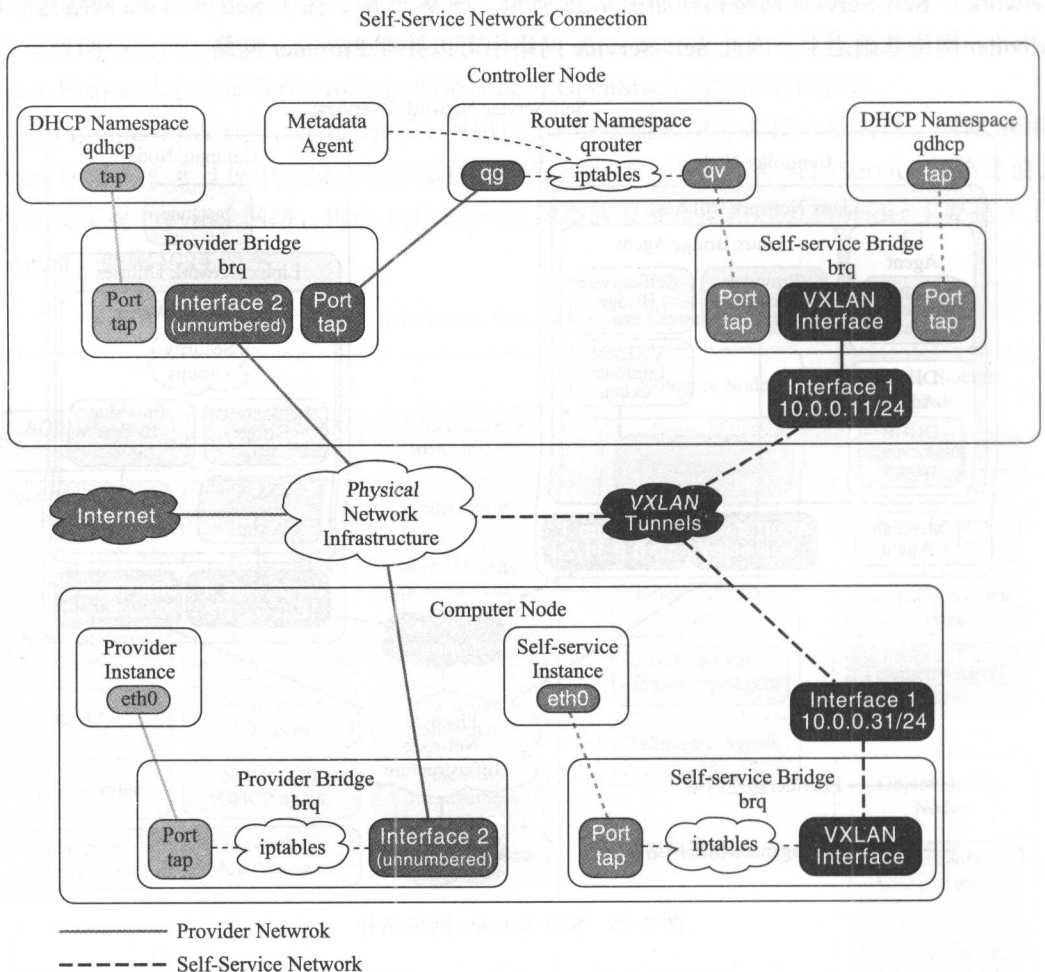


图 9-26 Self-Service 网络通信

Self-Service 网络通常使用的是 Overlay 网络技术，如 GRE 和 VXLAN。而 Overlay 网络协议包含了额外的数据包，这些额外的数据包不仅增加了网络传输的负载，还占用了用户数据的可用空间。因此，相对而言，Self-Service 网络在数据传输效率上不及 Provider 网

络,但是 Self-Service 网络的灵活可控性确实更适合云计算环境,尤其是基于 SDN 的网络技术实现。随着各种高性能的物理网卡和交换机的使用,加上 L2POP 等软件优化技术,Self-Service 网络与 Provider 网络在传输性能上的差距完全在可容忍范围之内,因此,Self-Service 网络才是 OpenStack 网络部署的主流。

9.4 Provider 网络部署与分析

Provider 网络(或者物理网络)是 Neutron 网络部署中一种拓扑简单、性能优异、可靠性较好的网络架构,但是 Neutron Provider 网络的这些优点是基于物理网络设备实现的,因此 Provider 网络本质上不具备云计算网络的弹性按需和灵活操作等特性。与其他的 Neutron 网络拓扑不同,Provider 网络下只有云管理员才能操作与 Neutron 网络相关的功能,因为 Provider 网络需要借助数据中心的物理网络才能实现,而普通的云终端租户无法接触到数据中心物理网络。同时,在 Provider 网络中并没有固定 IP 和浮动 IP 的概念,因为 Provider 拓扑只处理二层网络中的实例数据交换,三层以上的网络功能全由物理设备实现。对于很多熟悉传统 IT 架构中物理网络的管理员而言,通过 Provider 网络架构来实现 OpenStack 的网络功能是很简单和轻松的事情,因此,很多数据中心管理员都会采用 Provider 网络来测试和部署基于 OpenStack 的云计算环境。

在 Neutron 未实现分布式虚拟路由 DVR 之前(Juno 之前版本),OpenStack 集群中的网络,不论是租户网络之间的东西流向还是内外网之间的南北流向,全部都需要汇聚到网络节点才能进行网络通信,这也导致了 OpenStack 租户网络的性能和可靠性难以满足大规模生产集群的需求。在 Provider 网络中,网络节点的功能被物理网络设备接管,与租户网络中各种系统软件模拟实现的虚拟网络设备相比,Provider 网络中的物理网络设备性能更好和可靠性更高,因此,在早期的 OpenStack 集群部署中,很多用户都采用 Provider 网络结构来实现 OpenStack 网络功能。由于 Provider 网络仅实现了二层网络虚拟化,因此用户在部署 Provider 网络时只能采用 Flat 和 VLAN 方式进行不同网络的隔离,而无法使用 Vxlan 或 GRE 等基于封装协议的隧道网络^①。

9.4.1 Provider 网络基于 OpenvSwitch 实现

Neutron 的 Provider 网络可以通过各种开源或厂商的网络插件和代理来实现,在开源领域,使用最多的是 LinuxBridge 和 OpenvSwitch 网络插件和代理。由于 Havana 版本之后,LinuxBridge 和 OpenvSwitch 网络插件被 ML2 核心插件替代,因此目前的 Provider 网络部署中最常使用的便是 ML2 插件和 LinuxBridge 代理或 OpenvSwitch 代理。其中,ML2 插件通常部署在控制节点上,而 LinuxBridge 代理或 OpenvSwitch 代理部署在计算节点上,

① 更详细的讲解可以参考 Openstack 官方网站中的 Network Guide。

Provider 网络不需要网络节点。本节主要介绍基于 OpenvSwitch 开源网络技术的 Provider 网络部署实现，后一节将介绍基于 LinuxBridge 网络技术的 Provider 网络部署实现。为了便于介绍，这里以部署一个控制节点和两个计算节点组成的 VLAN 类型 Provider 网络为例，节点群集 Provider 网络拓扑如图 9-27 所示。

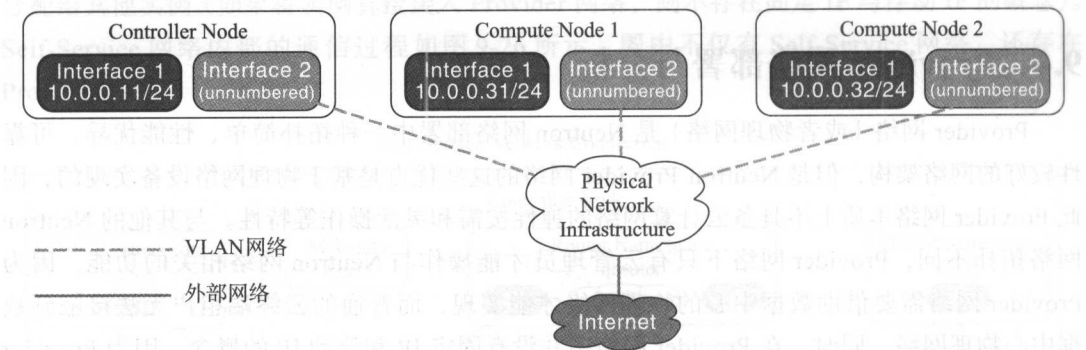


图 9-27 节点网络拓扑

图 9-27 所示中，控制节点和计算节点均至少需要两个网络接口，其中 Interface1 作为管理接口（管理网络为 10.0.0.0/24），Interface2 作为 Provider 接口。各个节点的 Provider 接口 Interface2 接入常规物理网络中（物理交换机或路由到外部的网络），Provider 网络接口上需要一个端口用于 OpenvSwitch 桥接，作为桥接用的物理网络端口不配置 IP。按照图 9-27 所示的网络拓扑，计算节点实例获取的 IP 属于 Provider 物理网络，因此实例数据经过计算节点内部 OpenvSwitch 虚拟机交换机之后，通过桥接直接进入物理网络，并通过物理交换机或路由与外网通信。

Neutron 网络服务在控制节点和两个计算节点上的分布如图 9-28 所示，其中，控制节点运行网络管理服务 Neutron-Server、ML2 插件、DHCP 代理和 OpenvSwitch 软件及其代理服务，计算节点除了运行 Nova-compute 服务之外，还需运行 ML2 插件、OpenvSwitch 软件及其代理服务。

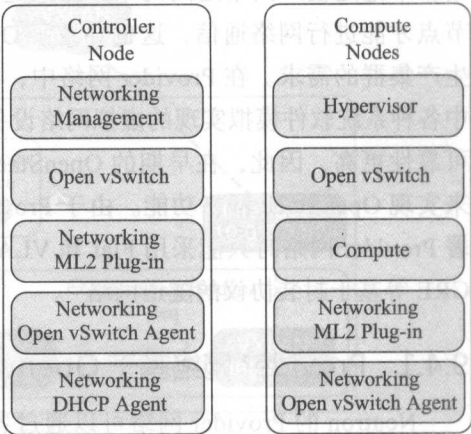


图 9-28 控制节点与计算节点节点服务布局

1. Provider 网络架构

Provider 网络的通用架构如图 9-29 所示，其中，Provider 网络架构使用物理网络设备进行二层网络数据的交换和三层数据的路由功能。

在 Provider 网络的常规架构中，控制节点主要包括 OpenvSwitch Agent 和 DHCP Agent 网络组件，其中，OpenvSwitch Agent 主要负责节点内部虚拟交换机的管理及其通信，同时

还负责通过虚拟端口与其他网络组件进行交互，如命名空间 Namespace 或其他底层的接口。DHCP Agent 主要负责管理 DHCP 命名空间，DHCP 命名空间主要负责为使用 Provider 网络的实例自动分发 IP。图 9-30 和图 9-31 是常规 Provider 网络架构下控制节点网络组件及组件之间的连接示意图。为了便于说明，图 9-31 所示中包含了两个不同的 Provider 网络。

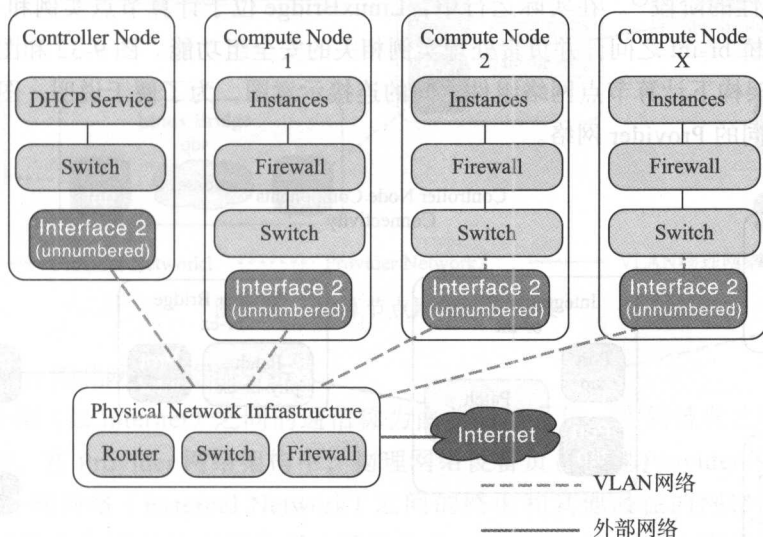


图 9-29 通用 Provider 网络架构

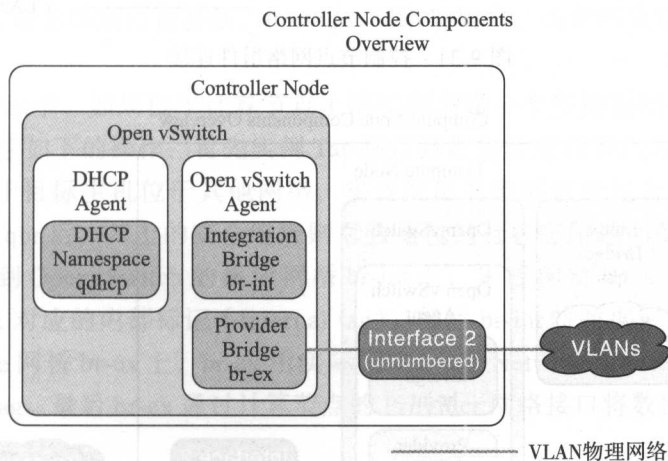


图 9-30 控制节点网络组件

与控制节点不同，计算节点主要运行的网络组件是 OpenvSwitch Agent 和 LinuxBridge Agent，其中 OpenvSwitch Agent 主要负责节点内部虚拟交换机的管理及其通信，同时还负责通过虚拟端口与其他网络组件的交互，如 LinuxBridge 或其他底层的网络接口。对于很多初次使用 OpenvSwitch 的用户可能会奇怪，既然使用的是 OpenvSwitch，为何还要需要

LinuxBridge? 其实, OpenvSwitch 部署中需要使用 LinuxBridge, 主要在于传统 LinuxBridge 对安全组 (SecurityGroups) 的处理要优于原生的 OVS 实现, 或者说 OpenvSwitch 需要借用 LinuxBridge 来进行安全组处理。相对 OpenvSwitch, LinuxBridge 要成熟稳定得多, 尽管 Native OpenvSwitch 也可以实现安全组功能, 但是对系统内核和 OVS 版本均有要求, 而且仍然处于实验性的阶段^①。在实际运行中, LinuxBridge 位于计算节点实例和 OpenvSwitch 创建的集成网桥 br-int 之间, 并负责处理实例相关的安全组功能。图 9-32 和图 9-33 是常规 Provider 网络架构下计算节点网络组件之间的连接示意图, 为了便于说明, 图 9-33 所示中包含了两个不同的 Provider 网络。

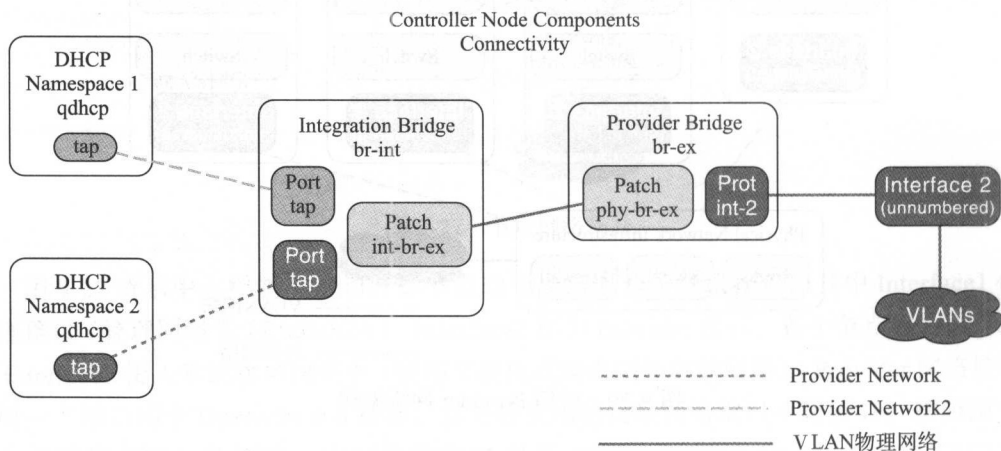


图 9-31 控制节点网络组件连接

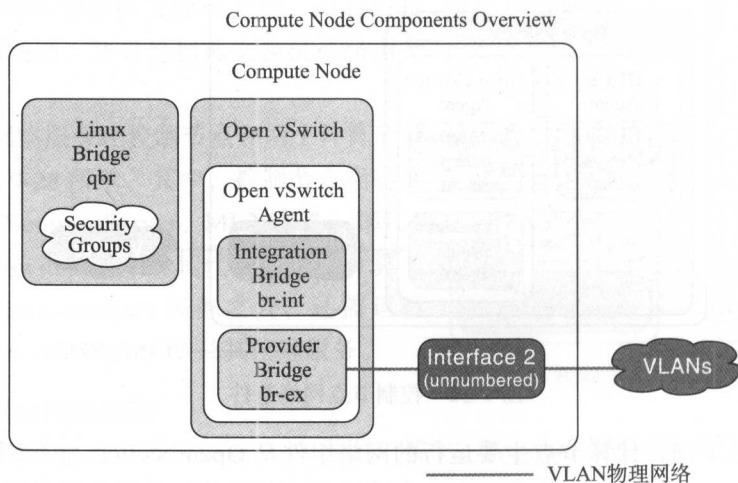


图 9-32 计算节点网络组件

① 参考 <http://docs.openstack.org/mitaka/networking-guide/config-ovsfdriver.html#config-ovsfdriver>。

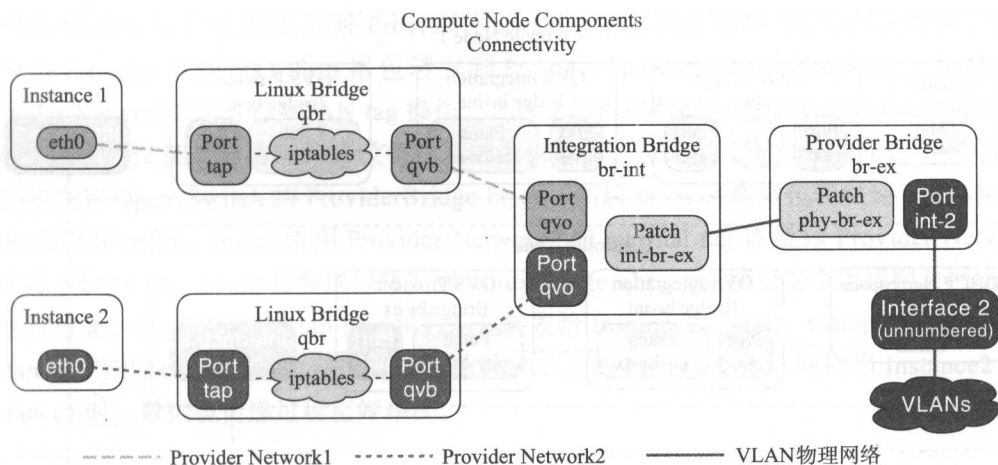


图 9-33 计算节点网络组件连接

2. Provider 南北网络数据流分析

实例与外网（如 Internet）之间的通信称为南北网络通信，实例彼此之间的通信称为东西网络通信。在 Provider 网络架构中，物理网络设备负责处理 Provider 网络（Provider Network）与外网网络（External Network）之间的路由和其他潜在的网络服务。为了便于说明，这里简单将可以供实例使用的网络称为 Provider Network，而将仅有路由才能访问的网络称为 External Network。而在实际使用中，Provider Network 是可以直接接入 External Network 而无须路由设备的。Provider 网络架构下，南北网络数据流向如图 9-34 所示。

在图 9-34 所示中，如果位于计算节点上的实例发送一个数据包到外网中的主机上，则计算节点将产生如下的操作：首先实例 Tap 接口将数据转发到节点内部的 LinuxBridge 网桥 qbr 上，由于目标主机位于其他网络，实例发出的数据包中包含目标主机的 MAC 地址，数据进入 qbr 后，其上的安全组规则对数据包进行状态跟踪和防火墙处理。之后 qbr 将数据转发到 OpenvSwitch 的集成网桥 br-int 上，集成网桥 br-int 为数据包添加与 Provider Network 对应的内部标记（Internal tag）。然后 br-int 将数据转发到 OpenvSwitch 的 ProviderBridge 网桥 br-ex 上，br-ex 用实际的 ProviderNetwork VLAN ID 替换掉 br-int 添加的 Internal tag。最后 br-ex 通过计算节点的 Provider 网络接口将数据包转发到物理网络设备上。

数据包进入物理网络设备后，将进行如下的操作：首先由交换机处理 Provider Network 与路由之间的 VLAN tag 操作；然后路由器将来自 Provider Network 的数据包路由到外部网络，交换机再次处理路由器与外网之间的 VLAN tag 操作；最后交换机将数据包转发到外网。外网主机对 Provider 网络中的实例访问过程与实例对外网主机的访问过程正好相反。

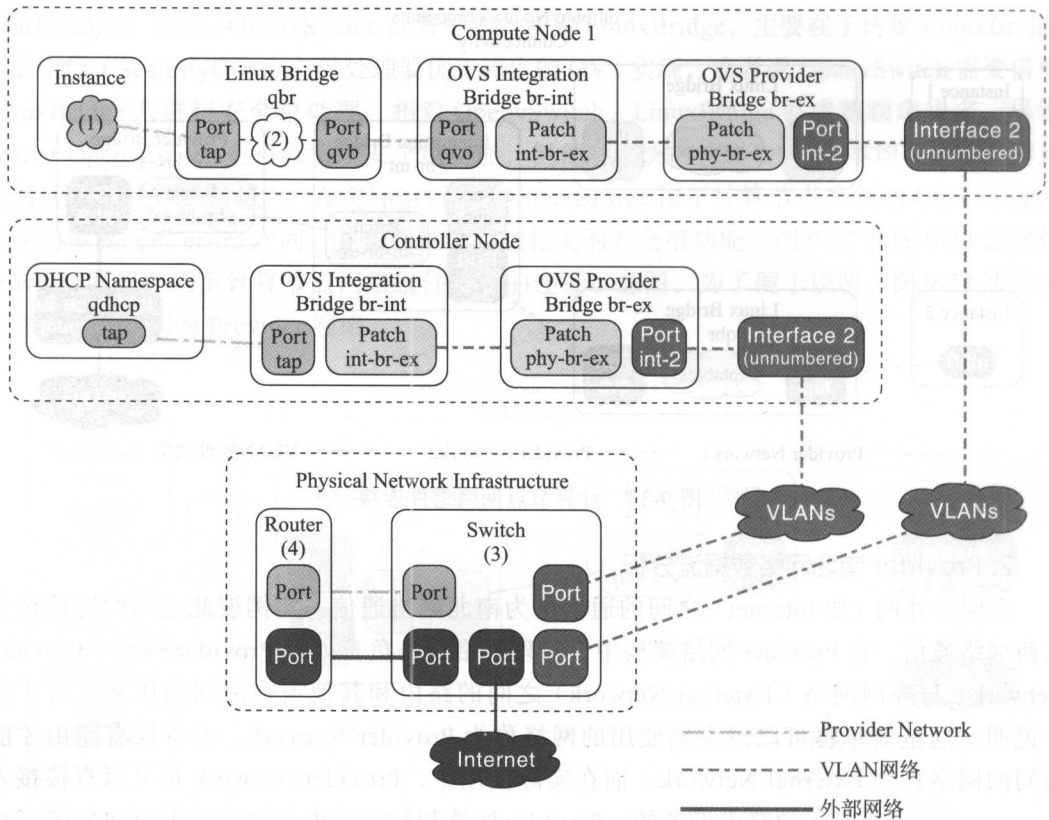


图 9-34 Provider 网络南北向数据流

3. Provider 东西数据流分析

Provider 网络内部实例之间的数据访问称为网络东西流向。Provider 网络东西数据流分为两种类型，即位于相同网段中的实例通信和不同网段之间的实例通信。二者的区别在于，不同 Provider 网段之间的实例通信需要具备三层路由功能的物理网络设备进行路由转发，而相同 Provider 网络中的实例通信只需二层物理交换机进行转发。不同 Provider 网络之间的实例通信过程如图 9-35 所示。

在图 9-35 中所示，当 Compute Node1 中的实例 Instance1 向 Compute Node2 中的实例 Instance2 发送数据包时，Compute Node1 中将会发生如下的数据传递操作。首先 Instance1 的 tap 接口将数据包转发到 LinuxBridge qbr，转发的数据包中包含了目标地址的 MAC 地址。qbr 中的安全组规则对数据包进行防火墙相关的操作，之后 qbr 将数据包转发到 OpenvSwitch 集成网桥 br-int。br-int 为数据包添加属于 Provider Network1 的 Internal Tag，然后 br-int 将数据包转发到 OpenvSwitch 的 ProviderBridge br-ex。br-ex 使用 Provider Network1 的实际 VLAN ID 替换 br-int 添加的 Internal tag。最后 br-ex 将数据包通过 Compute Node1 节点的 Provider 网络接口转发到物理网络设备中。物理网络设备接收到 Compute Node1 转

发来的数据包后，交换机处理 Provider Network1 与路由之间的 VLAN tag 操作，路由将来自 Provider Network1 的数据包转发到 Provider Network2，交换机再次处理路由与 Provider Network2 之间的 VLAN tag 操作，之后交换机将数据包转发到 Compute Node2 中。Compute Node2 接收到物理网络设备发送的数据包后，位于其上的 Provider 网络接口将数据包转发到 OpenvSwitch 的 ProviderBridge br-ex，然后 br-ex 将数据包转发到 OpenvSwitch 的集成网桥 br-int，br-int 使用 Provider Network2 的 internal tag 替换掉 Provider Network2 的实际 VLAN ID。br-int 将数据包转发到 LinuxBridge qbr，qbr 使用安全组规则对网络数据包进行过滤，之后 qbr 通过 tap 接口将数据包转发给 Instance2。至此，Compute Node1 上的 Instance1 发出的数据包成功到达 Compute Node2 上的 Instance2 中。同样当 Instance2 访问 Instance1 时，数据包传递过程正好相反。

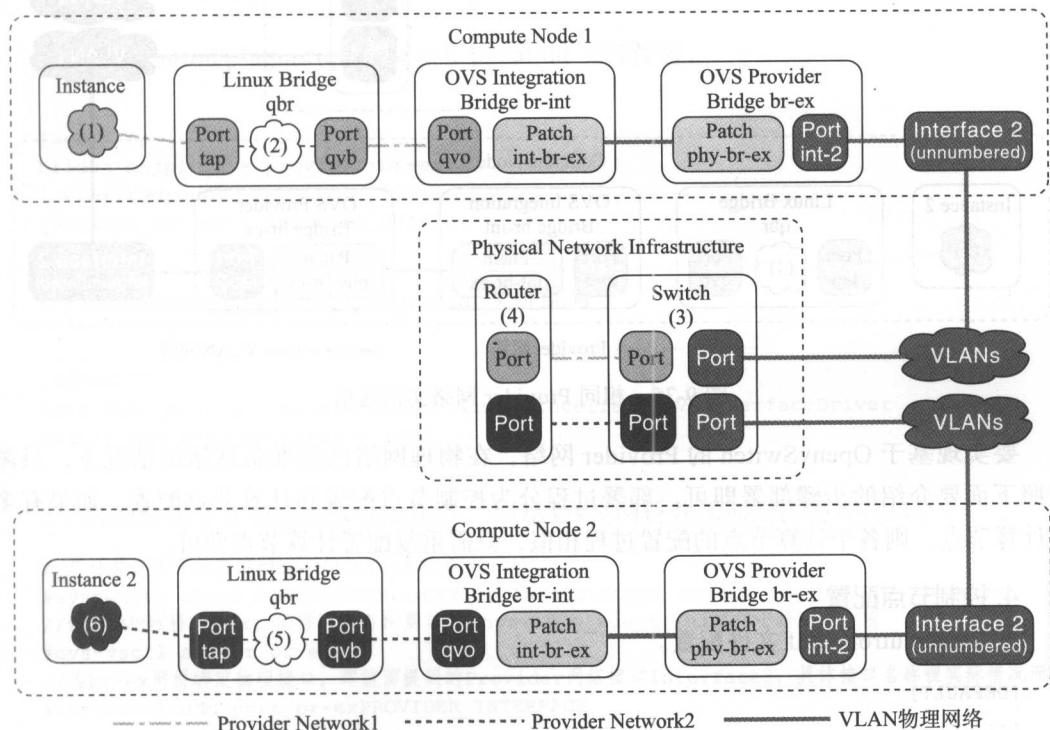


图 9-35 不同 Provider 网络实例通信

当 Compute Node1 中的 Instance1 与 Compute Node2 中的 Instance2 位于相同的 Provider 网络时，Instance1 对 Instance2 的访问过程与图 9-35 所示过程类似，但是由于相同网段通信不需路由转发，因此此时的网络设备无须具备三层路由功能，只需二层交换功能即可。相同 Provider 网络中的实例通信过程如图 9-36 所示。对比图 9-35 和图 9-36 可以看出，相对不同 Provider 网络中的实例通信，相同 Provider 网络中的实例通信过程除了无须物理设备进行路由功能外，数据流在 Compute Node1 和 Compute Node2 中的传递过程完全一样，而物理网络设备所要做的操

作只是简单地将来自 Compute Node1 中的数据以二层交换形式转发到 Compute Node2 即可。

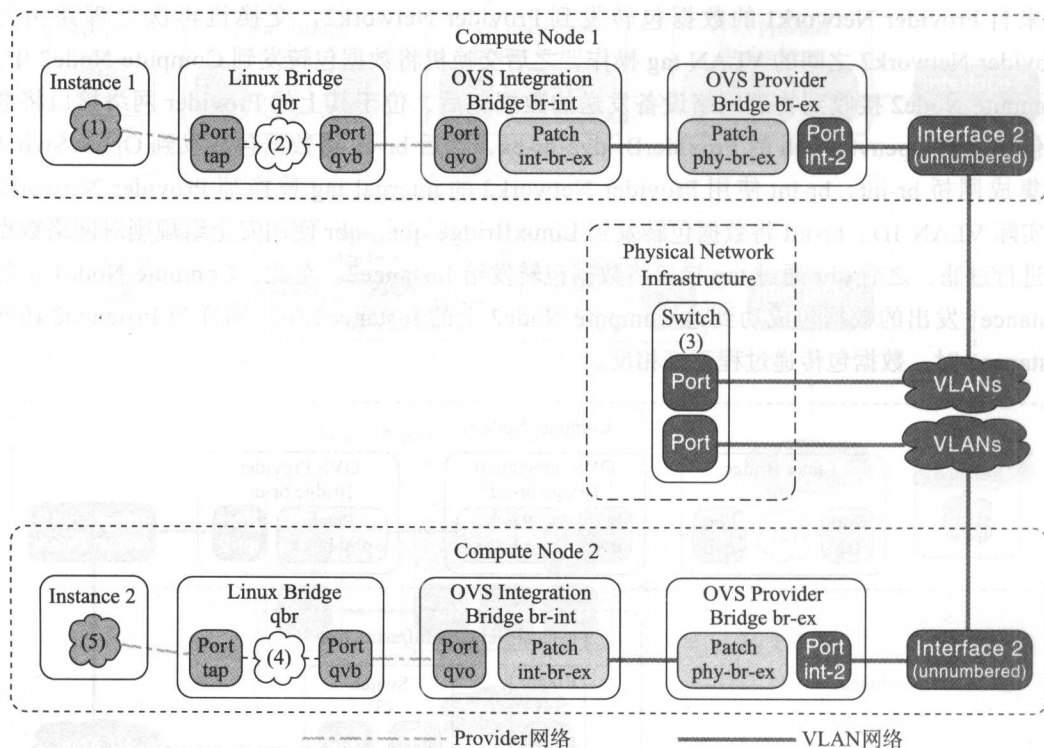


图 9-36 相同 Provider 网络实例通信

要实现基于 OpenvSwitch 的 Provider 网络，在物理网络已经准备就绪的情况下，只需按照下面要介绍的步骤部署即可。部署过程分为控制节点配置和计算节点配置，如果有多个计算节点，则各个计算节点的配置过程相似，只需重复配置计算节点即可。

4. 控制节点配置

1) /etc/neutron.conf 文件配置。

```
[DEFAULT]
.....
//使用ML2核心插件
core_plugin = ml2
//Provider网络不提供layer-3服务，如路由功能，因此服务插件留空
service_plugins =
.....
```

2) /etc/neutron/plugins/ml2/ml2_conf.ini 文件配置。

```
.....
//核心插件ML2配置
[ml2]
//ML2的TypeDriver列表，对于provider网络，只有Flat和VLAN两种
```

```

type_drivers = flat,vlan
//Provider网络架构不支持租户网络（私有网络），此处留空
tenant_network_types =
//ML2的MechanismDriver列表，这里使用的是OpenvSwitch
mechanism_drivers = openvswitch
extension_drivers = port_security
[ml2_type_flat]
//Flat网络的Label，这里设为Provider，在创建Flat类型的Provider网络时会使用此Label
flat_networks = provider
[ml2_type_vlan]
//VLAN网络的Label和VLAN ID范围，未指定ID范围则支持任意VLAN ID，这里仅有Label不指定ID范围
network_vlan_ranges = provider
[securitygroup]
//使用iptables_hybrid作为安全组策略
firewall_driver = iptables_hybrid
.....

```

3) /etc/Neutron/plugins/openvswitch_agent.ini 文件配置。

```

.....
[ovs]
bridge_mappings = provider:br-ex
[securitygroup]
firewall_driver = iptables_hybrid
.....

```

4) /etc/neutron/plugins/dhcp_agent.ini 文件配置。

```

.....
[DEFAULT]
interface_driver = Neutron.agent.linux.interface.OVSInterfaceDriver
enable_isolated_metadata = True
.....

```

5) 启动控制节点 OpenvSwitch 服务，创建 OVS 网桥并添加网桥物理接口。

```

//启动并设置开机自启动openvswitch服务
#systemctl start openvswitch.service && systemctl enable openvswitch.service
//创建OVS网桥br-ex，记得网桥名称要与openvswitch_agent.ini文件中指定的一致
#ovs-vsctl add-br br-ex
//为br-ex网桥绑定物理接口，即前面提到的Provider网络接口Interface2，具体接口名称视实际情况而定
#ovs-vsctl add-port br-exPROVIDER_INTERFACE

```

6) 启动控制节点上的网络服务。

```

//启动Neutron-Server服务
systemctl start neutron-server.service
systemctl enable neutron-server.service
//启动Neutron-Openvswitch-Agent服务
systemctl start neutron-openvswitch-agent.service
systemctl enable neutron-openvswitch-agent.service
//启动Neutron-Dhcp-Agent服务
systemctl start neutron-dhcp-agent.service
systemctl enable neutron-dhcp-agent.service

```

5. 计算节点配置

1) /etc/Neutron/plugins/openvswitch_agent.ini 文件配置。

```
[ovs]
bridge_mappings = provider:br-provider
[securitygroup]
firewall_driver = iptables_hybrid
```

2) 启动 OpenvSwitch 服务。

```
#systemctl start openvswitch.service && systemctl enable openvswitch.service
```

3) 创建 OVS 网桥，并添加 OVS 网桥物理接口。

```
//创建网桥br-ex
# ovs-vsctl add-br br-ex
//添加Provider网络接口，如eth2
# ovs-vsctl add-port br-exPROVIDER_INTERFACE
```

4) 启动 Neutron 网络服务。

```
# systemctl start neutron-openvswitch-agent.service
# systemctl enable neutron-openvswitch-agent.service
```

5) 在控制节点验证网络服务启动情况。

```
#neutron agent-list
+-----+-----+-----+-----+-----+
|          id          | agent_type          | host          | alive | admin_state_up |
+-----+-----+-----+-----+-----+
| ...801f068c57| Open vSwitch agent | controller    | :-)   | True            |
| ...de214fa303| DHCP agent         | controller    | :-)   | True            |
| ...0ca83a40cb| Open vSwitch agent | compute1      | :-)   | True            |
| ...44f0e4a24f| Open vSwitch agent | compute2      | :-)   | True            |
+-----+-----+-----+-----+-----+
```

至此，基于 OpenvSwitch 的 Provider 网络配置已经完成，并且各项网络服务已经在控制节点和计算节点成功启动，后续管理员即可基于已有物理网络进行 Provider 网络的创建。

9.4.2 Provider 网络基于 LinuxBridge 实现

基于 LinuxBridge 的 Provider 网络与基于 OpenvSwitch 的 Provider 网络具有类似的网络拓扑架构，只是由于具体网络技术的实现不同，在节点服务配置上需要做一些调整和设置。需要指出的是，OpenStack 默认的 ML2 插件使用的 MechanismDriver 是 OpenvSwitch，

而不是 LinuxBridge，但是在基于 OpenvSwitch 的网络模型中，同样会使用到 LinuxBridge，其主要原因在于 OpenvSwitch 在使用 Iptables 规则处理安全组时存在诸多限制，并且不如 LinuxBridge 成熟，因此在实例与 OpenvSwitch 的集成网桥 br-int 之间仍然需要 LinuxBridge 网桥 qbr 进行安全组相关功能的处理。相比 OpenvSwitch 较为复杂的网络处理过程，LinuxBridge 的网络处理功能更为简单，并且从发展历史来看，LinuxBridge 要更为成熟，因此很多基于 OpenStack 的网络方案都采用 LinuxBridge 来进行虚拟网络的实现，如 Rackspace 的私有云方案。

在基于 LinuxBridge 的 Provider 网络部署中，同样无须运行 L3 服务的网络节点，仅需控制节点和计算节点。由于 LinuxBridge 插件在 Havana 版本后被 ML2 核心插件替换，因此基于 LinuxBridge 的 Provider 网络实际上是由 ML2 插件和 LinuxBridge Agent 共同实现的，与基于 OpenvSwitch 的 Provider 网络服务布局不同，在使用 LinuxBridge 的 Provider 网络中，控制节点与计算节点的网络服务分布如图 9-37 所示。

控制节点主要运行的网络组件是 LinuxBridge Agent 和 DHCP Agent，其中 LinuxBridge Agent 主要负责虚拟交换机的管理及其连接，同时还负责通过虚拟端口与其他网络组件交互，如命名空间 Namespace 或其他底层的接口。DHCP Agent 主要负责管理 DHCP 命名空间，而 DHCP 命名空间主要负责为使用 Provider 网络的实例自动分发 IP。图 9-38 和图 9-39 分别是运行 LinuxBridge agent 的控制节点网络组件和网络组件之间的通信连接图。为了便于说明，图 9-39 所示中包含了两个不同的 Provider 网络。

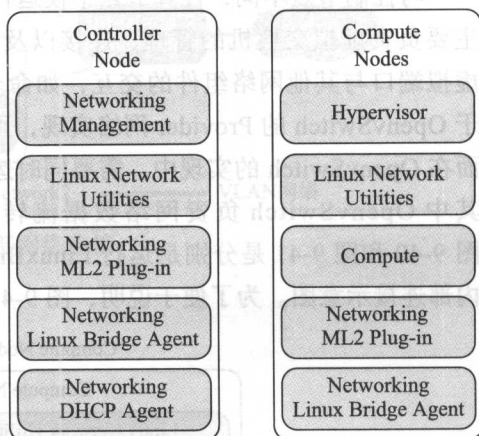


图 9-37 控制节点与计算节点网络服务布局

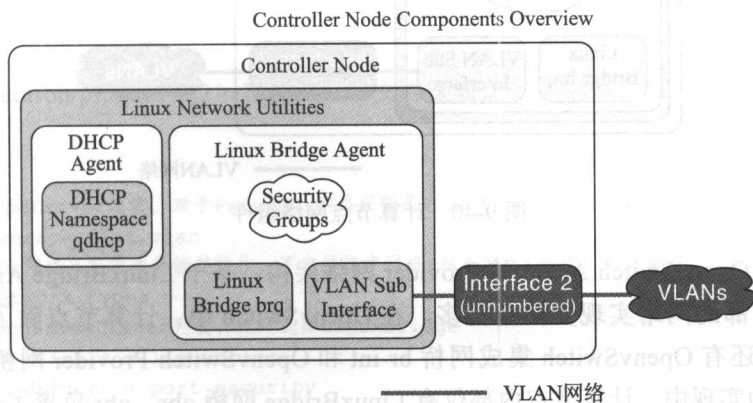


图 9-38 控制节点网络组件

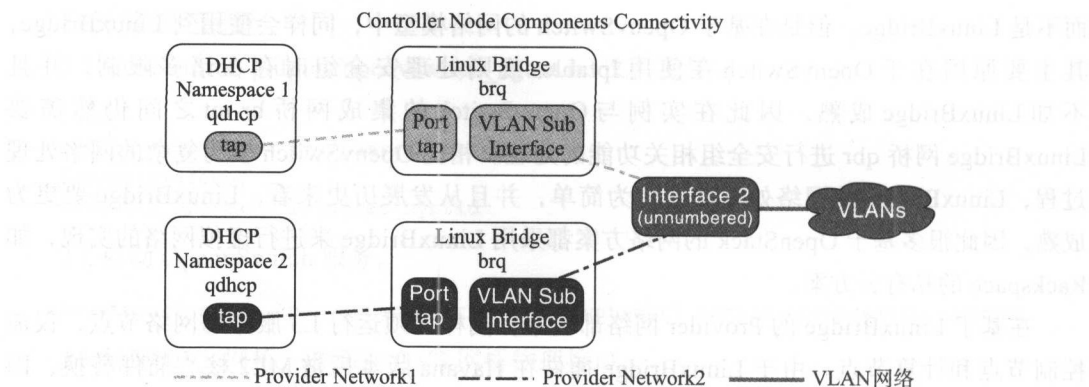


图 9-39 控制节点内部网络连接

与控制节点不同，计算节点中仅运行 LinuxBridge Agent 服务，并且 LinuxBridge Agent 主要负责虚拟交换机的管理、连接以及实例安全组 SecurityGroups 处理，同时还负责通过虚拟端口与其他网络组件的交互，如命名空间 Namespace 或其他底层的接口。通过对比基于 OpenvSwitch 的 Provider 网络实现，可以看到此处的计算节点仅运行 LinuxBridge Agent，而在 OpenvSwitch 的实现中，需要同时运行 OpenvSwitch Agent 和 LinuxBridge Agent 服务，其中 OpenvSwitch 负责网络数据流转发处理，而额外的 LinuxBridge 负责安全处理。图 9-40 和图 9-41 是分别是运行 LinuxBridge agent 的计算节点网络组件和网络组件之间的内部连接示意图。为了便于说明，图 9-41 所示中包含了两个不同的 Provider 网络。

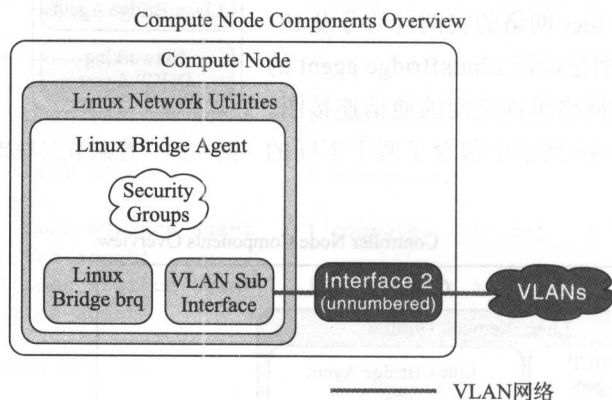


图 9-40 计算节点网络组件

对比基于 OpenvSwitch Agent 的 Provider 网络架构，基于 LinuxBridge Agent 的网络架构计算节点内部的网络实现要简单得多。在 OpenvSwitch 中，计算节点除了 LinuxBridge 网桥 qbr 外，还有 OpenvSwitch 集成网桥 br-int 和 OpenvSwitch Provider 网桥 br-ex。而在 LinuxBridge 的实现中，计算节点内部仅有 LinuxBridge 网桥 qbr，qbr 负责了全部数据相关的网络操作，因此基于 LinuxBridge 实现的 Provider 网络不论是在网络功能实现和网络故障

排查方面都要比基于 OpenvSwitch 的实现简单和轻松很多。同时由于仅运行 LinuxBridge，计算节点的运行的网络组件负载也要小得多。关于 LinuxBridge 实现的 Provider 网络南北和东西数据流分析，这里不再复述，其过程与 OpenvSwitch 实现的网络相似，只是没有了集成网桥 br-int 和 Provider 网桥 br-ex，具体的分析过程可以参考 OpenStack 官方网站的网络指南部分。下面介绍基于 LinuxBridge 的 Provider 网络部署过程。与 OpenvSwitch 一样，部署过程也分为控制节点和计算节点部署，并且全部计算节点的部署过程完全一样，用户只需重复对计算节点进行配置即可。

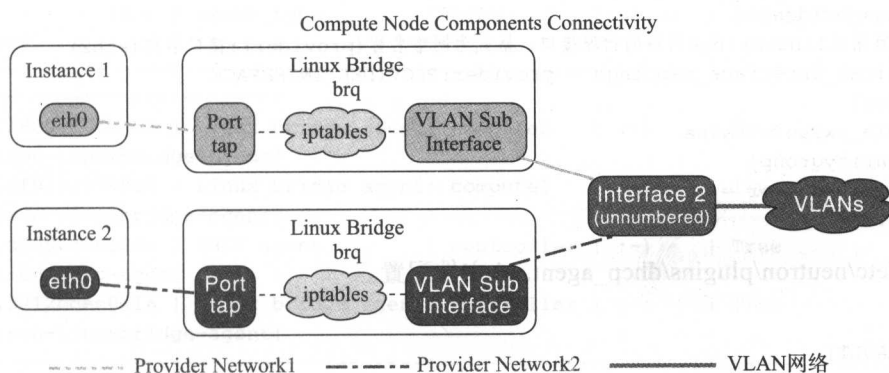


图 9-41 计算节点内部网络连接

1. 控制节点配置

1) /etc/Neutron.conf 文件配置。

```
.....
[DEFAULT]
//使用ML2核心插件
core_plugin = ml2
// Provider网络不提供L3等高级网络服务，服务插件留空
service_plugins =
.....
```

2) /etc/neutron/plugins/ml2/ml2_conf.ini 文件配置。

```
.....
[ml2]
//ML2的TypeDriver列表，对于Provider网络只能是Flat和VLAN
type_drivers = flat,vlan
//Provider网络只有管理员能够操作，不支持租户网络(私有网络)创建，此处留空
tenant_network_types =
//ML2的MechanismDriver列表，此处选择LinuxBridge
mechanism_drivers = linuxbridge
extension_drivers = port_security
[ml2_type_flat]
//Flat类型Provider网络的Label，这里设为Provider，如果创建Flat网络则需要用到此Label
```

```
flat_networks = provider
[m12_type_vlan]
//VLAN类型Provider网络的Label和VLAN ID设置, 未指定IP范围表示支持全部的VLAN ID
network_vlan_ranges = provider
[securitygroup]
firewall_driver = iptables
.....
```

3) /etc/neutron/plugins/linuxbridge_agent.ini 文件配置。

```
.....
[linux_bridge]
//设置用以linuxbridge网桥的物理接口, 格式为网络名称(provider):接口名称(eth2)
physical_interface_mappings = provider:PROVIDER_INTERFACE
[vxlan]
enable_vxlan = False
[securitygroup]
firewall_driver = iptables
.....
```

4) /etc/neutron/plugins/dhcp_agent.ini 文件配置。

```
.....
[DEFAULT]
interface_driver = Neutron.agent.linux.interface.OVSInterfaceDriver
enable_isolated_metadata = True
.....
```

5) 启动控制节点网络服务。

```
//启动neutron-server服务
systemctl start neutron-server.service
systemctl enable neutron-server.service
//启动neutron-linuxbridge-agent服务
systemctl start neutron-linuxbridge-agent.service
systemctl enable neutron-linuxbridge-agent.service
//启动neutron-dhcp-agent服务
systemctl start neutron-dhcp-agent.service
systemctl enable neutron-dhcp-agent.service
```

2. 计算节点配置

1) /etc/neutron/plugins/linuxbridge_agent.ini 文件配置

```
[linux_bridge]
//设置LinuxBridge网桥的物理接口, 格式为网络(provider):接口名称(eth2)
physical_interface_mappings = provider:PROVIDER_INTERFACE
[vxlan]
enable_vxlan = False
[securitygroup]
firewall_driver = iptables
```

2) 启动计算节点网络服务。

```
//启动neutron-linuxbridge-agent服务
systemctl start neutron-linuxbridge-agent.service
systemctl enable neutron-linuxbridge-agent.service
```

3) 验证计算节点网络服务运行情况。

```
# neutron agent-list
+-----+-----+-----+-----+-----+
|          id          | agent_type          | host          | alive | admin_state_up |
+-----+-----+-----+-----+-----+
|...18801f068c57 | Linux bridge agent| compute2      | :-)   | True            |
neutron-linuxbridge-agent|
|...df924e0788a4 | Linux bridge agent| compute1      | :-)   | True            |
neutron-linuxbridge-agent|
|...d689630b629e | DHCP agent         | controller    | :-)   | True            |
neutron-dhcp-agent      |
|...3712bdec0416 | Linux bridge agent| controller    | :-)   | True            |
neutron-linuxbridge-agent|
+-----+-----+-----+-----+-----+
```

从配置过程来看, 基于 LinuxBridge 的 Provider 网络配置比基于 OpenvSwitch 的 Provider 网络要简单很多, 前者由于无须运行 OpenvSwitch, 因此也无须手工创建 OVS 网桥, 进而也无须手工为 OVS 网桥绑定网络接口, 而仅需在 ML2 的配置文件中指定接入 Provider 网络的物理网口。至此, 基于 LinuxBridge 的 Provider 网络已经配置完成, 之后管理员便可创建 Provider 网络, 而不论是基于 OpenvSwitch 还是 LinuxBridge 的 Provider 网络, 其创建过程都一样, 只是底层的网络实现机制不同而已。

9.4.3 Provider 网络创建与验证

前面两节介绍了基于 OpenvSwitch 和 LinuxBridge 的 Provider 网络部署, 本节将介绍用户如何创建 VLAN 类型的 Provider 网络。VLAN 类型的 Provider 网络在创建时需要指定具体的物理网络 VLAN ID, 因为实例直接接入 VLAN 物理网络, 因此从网络拓扑层面看, 虚拟机就如同在 VLAN 网络中的物理机一样, 都是通过 VLAN ID 来进行网络隔离和通信, 并且不存在如 GRE 或 VxLAN 这样的 Overlay 网络。在创建 VLAN 类型的 Provider 网络之前, 管理员需要事先规划该 Provider 网络应该使用的 VLAN ID, 因为创建完成后, 接入此 Provider 网络的实例便属于此 VLAN。Provider 网络的创建语法如下:

```
//VLAN类型
neutron net-create --provider:physical_network=<provider label>\
--provider:network_type=vlan --provider:segmentation_id=<vlan id>
```

```
[--router:external True][--share] <network name>
//Flat类型
neutron net-create --provider:physical_network=<provider label>\
--provider:network_type=flat[--share] [--router:external True] <network name>
```

Neutron 在创建 Provider 网络时，需要指定 Provider Attributes，如 --provider:physical_network、--provider:network_type 和 --provider:segmentation_id。其中，provider:physical_network 用以指定 Provider 物理网络的名称（PHYSICAL_NAME），其又称为 Provider 网络的 Label，该值由用户在配置文件 /etc/neutron/plugins/ml2/ml2_conf.ini 中设置，具体如下：

```
//设置Flat类型网络的Label(或物理网络名称physical_name)
[ml2_type_flat]
flat_networks = provider
//设置VLAN类型网络的Label，此处可以指定VLAN ID范围(如n:m)；也可以不指定，不指定则ID不受限制
[ml2_type_vlan]
network_vlan_ranges = provider:n:m
```

另外一个 Provider 属性 provider:network_type 用以指定用户创建的网络拓扑类型。对于 Provider 网络，用户可以选择 Flat 和 VLAN，而对于 Self-Service 网络，还可以是 GRE、VxLAN 和 Local。而属性 provider:segmentation_id 仅对 VLAN 类型的 Provider 网络有效，即分配给所创建 VLAN 类型 Provider 网络的 VLAN ID，这必须是真实存于物理网络中的 VLAN ID，否则实例不能通信。此外，还有两个可选参数，即 --router:external True 和 --share。router:external True 表示用户创建的 Provider 网络不直接接入外部网络，而是通过 Router 接入；Share 则表示其他用户也可以使用当前用户创建的 Provider 网络。关于 provider:physical_network、provider:network_type 和 provider:segmentation_id 三个网络属性在创建不同类型网络时的取值情况，可以参考表 9-4。

表 9-4 Neutron 网络创建时网络属性设定

provider:network_type	provider:physical_network	provider:segmentation_id
flat	必须设置，flat 网络需要提供物理网络信息	NULL，Flat 网络没有网络标记
vlan	必须设置，vlan 网络需要提供物理网络信息	必须设置，VLAN 网络使用的 VLAN ID
gre/vxlan	NULL，gre/vxlan 网络无须物理网络信息	必须设置，gre/vxlan 网络使用的 Tunnel ID
local	NULL，local 网络无须物理网络信息	NULL，local 网络无须网络标记

另外，在 Neutron 网络的创建过程中，有些用户可能想知道网络节点和计算节点上的 Provider 网络物理接口是如何绑定到网桥的，是通过配置文件自动设置还是需要手动设置？这里需要根据使用的网络插件和代理来具体分析。以 OpenvSwitch 和 LinuxBridge 为例：

1. OpenvSwitch

控制节点 ml2_conf.ini 配置文件中有如下设置：

```
[ml2_type_flat]
flat_networks = provider      //Flat物理网络名称
[ml2_type_vlan]
```



```
network_vlan_ranges = provider //VLAN物理网络名称
```

控制节点和计算节点 `openvswitch_agent.ini` 配置文件中如下配置：

```
[ovs]
```

```
bridge_mappings = provider:br-ex //此处的物理网络名称Provider要对应到ml2_conf.ini文件
```

控制节点和计算节点中，在启动 `OpenvSwitch` 服务后和启动 `Neutron` 网络服务前，手工执行如下命令：

```
# ovs-vsctl add-br br-ex //此处的br-ex要对应到openvswitch_agent.ini文件
# ovs-vsctl add-port br-ex eth2 //此处的eth2即是接入Provider网络的节点物理接口
```

完成上述配置和命令，启动 `Neutron` 服务后，对于 `OpenvSwitch` 下的 `Provider` 网络，控制节点和计算节点上的物理网络网口 `eth2` 便绑定到了 `OpenvSwitch` 的 `br-ex` 网桥中。

2. LinuxBridge

`LinuxBridge` 要简单得多，`LinuxBridge` 无须手工创建网桥并为其添加物理网口，而是在 `Provider` 网络创建时自动配置。控制节点 `ml2_conf.ini` 配置文件中如下设置：

```
[ml2_type_flat]
flat_networks = provider //Flat物理网络名称
[ml2_type_vlan]
network_vlan_ranges = provider //VLAN物理网络名称
```

控制节点和计算节点 `linuxbridge_agent.ini` 配置文件中如下配置：

```
[linux_bridge]
physical_interface_mappings = provider:eth2 //此处的物理网络名称provider要对应到
ml2_conf.ini
```

在用户创建基于 `LinuxBridge` 的 `Provider` 网络时，基于上述两个配置文件，控制节点和计算节点的物理网口 `eth2` 便会自动添加到 `LinuxBridge` 网桥上。

3. Provider 网络创建

现在根据 `Neutron` 的 `Provider` 网络创建语法。假设已有 `VLAN ID` 为 10 的物理网络，并且物理网络名称在 `ml2_conf.ini` 配置文件中已被设置为 `Provider`，则可以按照如下步骤创建 `VLAN ID` 为 10 的 `Provider` 网络并进行验证。

1) 创建一个名为 `Provider_10` 的 `VLAN` 类型 `Provider` 网络。

```
[root@mitaka ~]# neutron net-create --shared --provider:physical_network provider\
--provider:network_type vlan --provider:segmentation_id 10 provider_10
[root@mitaka ~]# neutron net-list
```

```
+-----+-----+-----+
| id                  | name          | subnets |
+-----+-----+-----+
| 68fafb57-b435-4d96-b5e7-56a1ef99cd00 | Provider_10 |          |
+-----+-----+-----+
```

2) 创建 `Provider` 网络 `Provider_10` 的子网 `provider_10-subnet`。

```
[root@mitaka ~]# neutron subnet-create provider_10 192.168.115.0/24 --name
provider_10-subnet --gateway 192.168.115.254
[root@mitaka ~]# neutron net-list
```

id	name	subnets
68fafb57-b435-4d96-b5e7-56a1ef99cd00	Provider_10	885270c5-32cb-4fd3-8f17-df2fc198e1ac 192.168.115.0/24

4. Provider 网络验证

1) 在控制节点上验证 DHCP 命名空间已经被创建。

```
[root@mitaka ~]# ip netns
qdhcp-68fafb57-b435-4d96-b5e7-56a1ef99cd00 (id: 0)
```

2) 创建安全组并添加适当的规则，以便对实例进行 SSH 和 ping 操作。

```
[root@mitaka ~]# nova secgroup-add-rule default icmp -1 -1 0.0.0.0/0
```

IP Protocol	From Port	To Port	IP Range	Source Group
icmp	-1	-1	0.0.0.0/0	

```
[root@mitaka ~]# nova secgroup-add-rule default tcp 22 22 0.0.0.0/0
```

IP Protocol	From Port	To Port	IP Range	Source Group
tcp	22	22	0.0.0.0/0	

3) 在 provider_10 网络上创建一个测试实例。

```
[root@mitaka ~]# nova image-list
```

ID	Name	Status	Server
1312da38-d043-4ce2-a011-3ea781ee8419	cirros-0.3.4-x86_64	ACTIVE	

```
[root@mitaka ~]# nova keypair-list
```

Name	Type	Fingerprint
mykey	ssh	a9:94:89:7b:ce:3f:b5:13:57:a8:a6:1d:72:10:f9:b4

```
[root@mitaka ~]# nova boot --image cirros-0.3.4-x86_64 --flavor 1 testserver
```

```
[root@mitaka ~]# nova list
```

ID	Name	Status	Task State	Power State	Networks
...27ef4f0af93c 192.168.115.2	testserver	ACTIVE	-	Running	Provider_10=

4) 在控制节点或其他 VLAN ID 为 10 的主机上 ping 实例 testserver。

```
$ ping -c 4 192.168.115.2
PING 192.168.115.2 (192.168.115.2) 56(84) bytes of data.
64 bytes from 192.168.115.2: icmp_req=1 ttl=63 time=3.18 ms
64 bytes from 192.168.115.2: icmp_req=2 ttl=63 time=0.981 ms
64 bytes from 192.168.115.2: icmp_req=3 ttl=63 time=1.06 ms
64 bytes from 192.168.115.2: icmp_req=4 ttl=63 time=0.929 ms
```

5) 访问 testserver 实例，并测试 ping Internet 外网。

```
$ ping -c 4 openstack.org
PING openstack.org (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_req=1 ttl=53 time=17.4 ms
64 bytes from 8.8.8.8: icmp_req=2 ttl=53 time=17.5 ms
64 bytes from 8.8.8.8: icmp_req=3 ttl=53 time=17.7 ms
64 bytes from 8.8.8.8: icmp_req=4 ttl=53 time=17.5 ms
```

到此，VLAN 类型的 Provider 网络已成功创建，并且用户实例可以通过 VLAN 网络与外网相互访问。由于管理员创建的是共享模式的 Provider 网络，因此租户可以直接使用此 Provider 网络为自己的实例提供网络通信服务。

9.5 Self-Service 网络部署与高可用

9.5.1 Self-Service 网络实现

Self-Service 网络又称租户网络，在实现 Self-Service 网络之前，管理员必须已经实现 Provider 网络，因此在 Self-Service 网络中，用户也可以直接使用 Provider 网络，并将实例直接接入 Provider 物理网络而不是租户私有云网络。也可以认为 Self-Service 网络是对 Provider 网络的扩展和增强实现。与 Provider 网络相比，Self-Service 最大的不同在于租户可以按需创建自己的私有网络，并且网络中需要提供 L3 服务以实现网络的东西和南北数据流。此外，从主流的 Self-Service 网络与 Provider 网络的部署节点上看，Self-Service 网络的 Neutron 服务组件通常分布在控制节点、网络节点和计算节点上，而 Provider 网络并不需要网络节点。由于 Self-Service 网络是基于 Provider 网络实现的，因此 Self-Service 网络也可以采用 OpenvSwitch 和 LinuxBridge 等开源网络插件或其他商业网络插件来实现。由于

OpenvSwitch 为 OpenStack 官方默认的网络插件，因此本节将主要介绍基于 OpenvSwitch 的 Self-Service 网络实现；另外一种基于 LinuxBridge 的 Self-Service 网络实现可以参考 OpenStack 官方文档中的网络指南部分。

与基于 OpenvSwitch 的 Provider 网络类似，Self-Service 网络的网络插件仍然选择 ML2 核心插件，而不是 OpenVSwitch Plugin（此插件已被 ML2 替换），同时代理选择 OpenvSwitch Agent，而 ML2 的 MechanismDriver 选择 OVS（即 OpenvSwitch），如果想要基于 LinuxBridge 实现 Self-Service 网络，则 MechanismDriver 选择 linuxbridge。Self-Service 网络要注意的几个网络概念是 Project (Tenant) 网络、外部网络 (External) 和路由 (Router)。

- ❑ Project 网络：Project 网络是 Self-Service 特有的网络。Project 网络提供特定 Project 中的实例网络连接，即特定 Project 的实例仅接入所属的 Project 网络。常规租户可以在云管理员的定义范围内管理自己的 Project 网络，根据管理员设置的外部网络 (External 网络) 类型的不同，Project 网络可以使用 VLAN、GRE 和 VxLAN 网络类型进行数据传输。Project 网络通常又称租户私有网络，因此不能直接访问其他外部网络（如 Internet 等），而 Project 实例从 Project 网络中获取的 IP 通常称为固定 IP (Fixed IP)。
- ❑ External 网络：External 网络其实就是 Provider 网络。External 网络提供了如 Internet 等外部网络的连接，由于 External 网络涉及与数据中心物理网络设备的交互，因此仅有经过授权的管理员才能管理 External 网络。根据数据中心物理网络设备的功能特性，External 网络可以是 Flat 或者 VLAN 类型的网络。由于 Flat 网络本质上并未进行网络标记，因此每个外部桥接仅能存在一个 Flat 网络，或者说每张物理网卡仅能存在一个 Flat 网络。如果存在多个不同网段而仅有一个外部桥接，则 External 网络只能使用 VLAN，否则需要增加多张网卡并配置多个外部桥接才能实现多个 Flat 网络以进行网络隔离。通常如果管理员创建的 External 网络为 Flat 类型，则 Project 网络选用 VxLAN/GRE 类型；而如果 External 网络为 VLAN 类型，则 Project 网络也选用 VLAN 类型。管理员为 External 网络分配的 IP 通常称为 Public IP。如果实例需要绑定 Floating IP，则 Floating IP 从 External 网络中分配。
- ❑ Router：通常 Router 仅在 Self-Service 网络中才会使用，Provider 网络中使用物理设备实现路由功能。Router 主要负责连接 Project 网络和 External 网络，Router 的 SNAT 功能实现了 Project 网络中实例对外部网络（如 Internet）的访问，管理员需要为每个 Router 分配一个 External 网络的 IP 以实现 SNAT。Router 的 DNAT 功能实现了外部网络（如 Internet）对 Project 网络中实例的直接访问，为外部网络提供了对 Project 网络中实例访问的路由 IP（称为 Floating IP）。此外，Router 还实现了相同 Project 中 Project 网络的连接。因此，Self-Service 网络中的 Router 同时提供了南北和东西的数据通信。

在最简单的三节点 Self-Service 网络部署中，为了 Project 网络既可以使用 VLAN 类

型,也可以使用 GRE/VxLAN 类型,控制节点至少需要一个网络接口(管理网络),网络节点至少需要四个网络接口(管理网络、隧道网络、VLAN 网络和外部网络),计算节点至少需要三个网络接口(管理网络、隧道网络和 VLAN 网络)。需要指出的是,如果 External 网络和 Project 网络都采用 VLAN 类型,则节点互联的物理网络设备必须支持 VLAN tagging。Self-Service 网络的节点 Neutron 组件部署和常规架构如图 9-42 和图 9-43 所示。

Service Layout

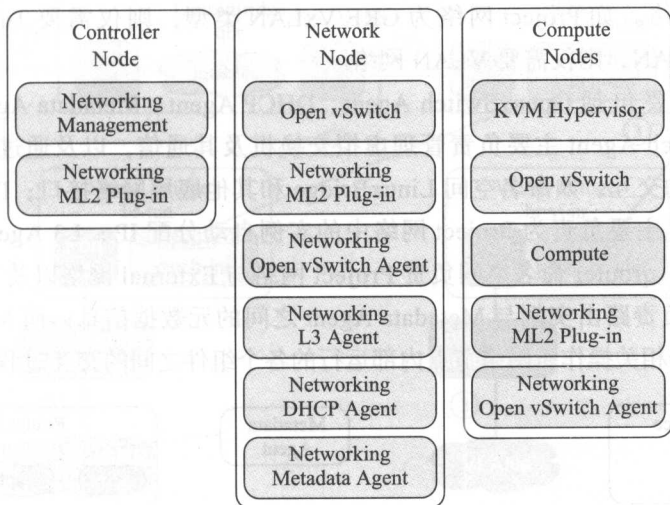


图 9-42 Self-Service 网络中的节点 Neutron 服务部署

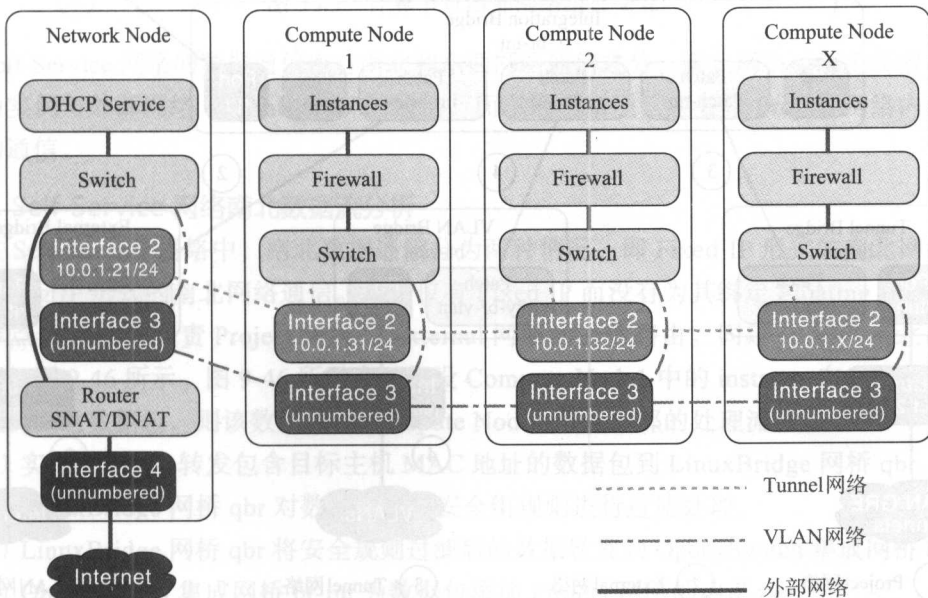


图 9-43 Self-Service 网络架构

图 9-42 所示中, 计算节点上除了运行 Nova-compute 服务外, 还运行 Neutron 的 ML2 插件和 OpenvSwitch 软件及其代理 Agent, 控制节点主要运行 Neutron-Server 和 ML2 插件, 网络节点运行 OpenvSwitch 软件及其 Agent、ML2 插件、L3 Agent、DHCP Agent 和 Metadata Agent。图 9-43 所示中, 每个计算节点都有 VLAN 网络和 Tunnel 网络, 而网络节点包括 VLAN 网络、Tunnel 网络和外部网络。需要指出的是, 计算节点和网络节点上并非同时需要 VLAN 网络和 Tunnel 网络。根据 Project 网络类型, 用户也可以只部署 VLAN 网络或 Tunnel 网络。如 Project 网络为 GRE/VxLAN 类型, 则仅需要 Tunnel 网络, 如果 Project 网络为 VLAN, 则仅需要 VLAN 网络。

网络节点主要包括 OpenvSwitch Agent、DHCP Agent、Metadata Agent 和 L3 Agent。其中, OpenvSwitch Agent 主要负责管理虚拟交换机及其通信, 以及通过虚拟交换机端口与其他网络组件的交互, 如命名空间 LinuxBridge 和其他底层网络接口; DHCP Agent 管理 DHCP 命名空间, 主要负责为 Project 网络中的实例自动分配 IP; L3 Agent 主要负责管理 qrouter 命名空间; qrouter 命名空间负责 Project 网络与 External 网络以及 Project 网络之间的连接, 同时还负责路由实例与 Metadata Agent 之间的元数据信息; 而 Metadata Agent 主要负责实例元数据相关操作。网络节点内部运行的各个组件之间的交互过程如图 9-44 所示。

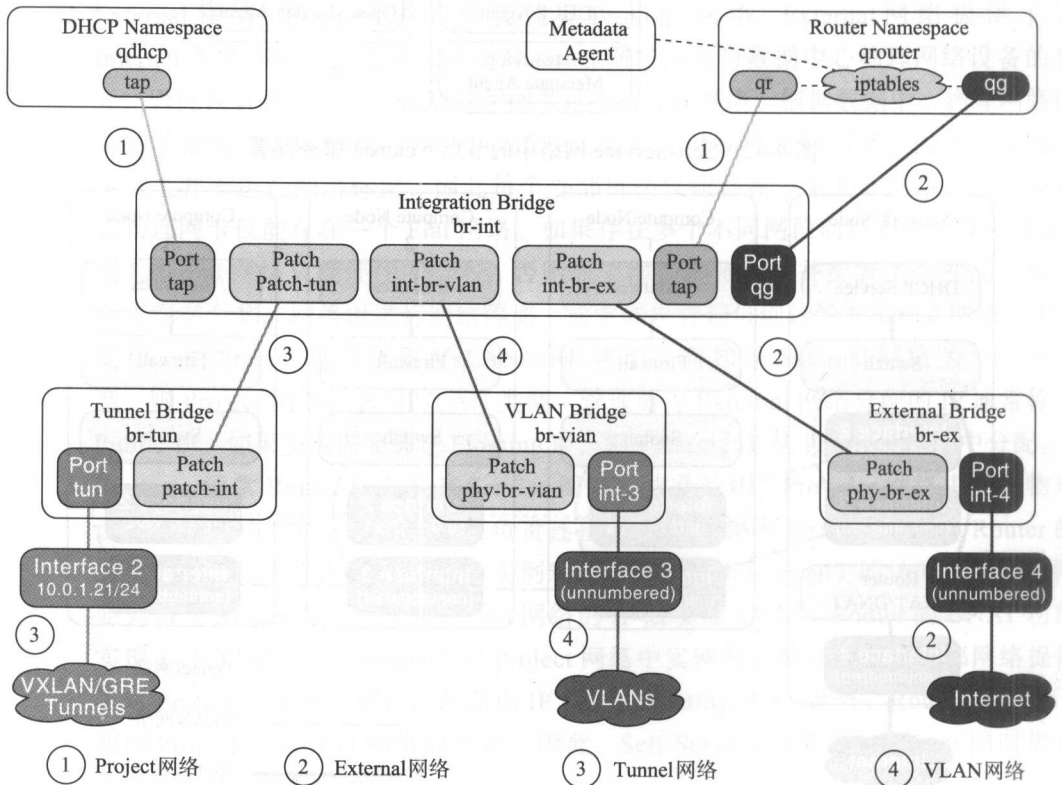


图 9-44 网络节点内部各个组件交互通信

计算节点主要运行的网络组件是 OpenvSwitch Agent 和 LinuxBridge Agent。其中 OpenvSwitch Agent 主要负责节点内部虚拟交换机的管理及其通信，同时还负责通过虚拟交换机端口与其他网络组件的交互，如 LinuxBridge 或其他底层的网络接口；而 LinuxBridge Agent 主要用于处理与实例相关的安全组问题。计算节点内部各个网络组件之间的交互过程如图 9-45 所示。

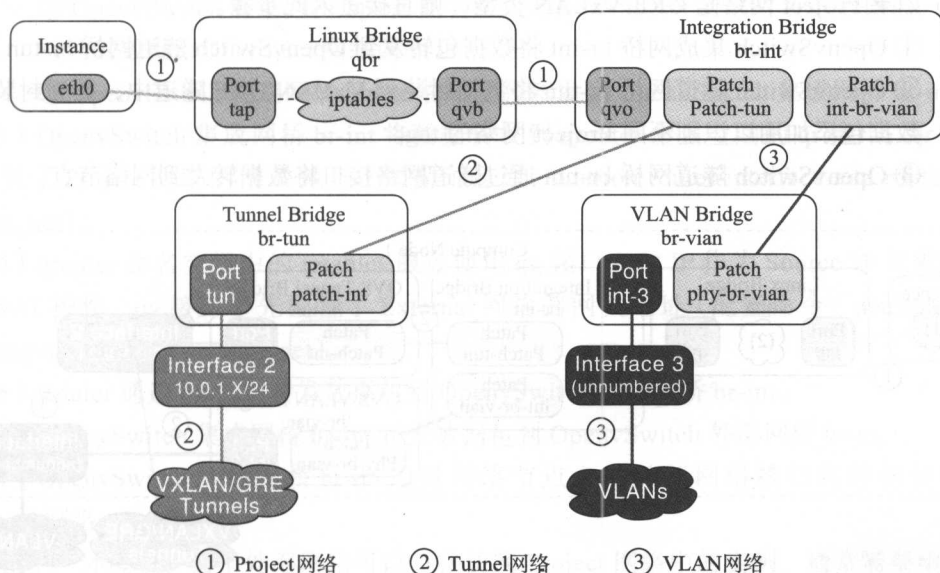


图 9-45 计算节点内部网络组件交互

Self-Service 网络的数据通信也有南北和东西网络通信之分，南北网络通信主要指 Project 网络内实例与外部网络（如 Internet）的通信，而东西网络通信主要指 Project 网络内部实例之间的通信。

1. Self-Service 网络南北数据流分析

在 Self-Service 网络中，南北数据通信分为两种情况，即 Fixed IP 形式的南北网络通信和 Floating IP 形式的南北网络通信。对于仅有 Fixed IP 而没有为其绑定 Floating IP 的实例，网络节点的 Router 负责 Project 网络与 External 网络的通信路由。固定 IP 实例南北网络通信过程如图 9-46 所示。图 9-46 所示中，假设 Compute Node1 中的 instance 向 External 网络中的主机发送数据包，则该数据包在 Compute Node1 节点内部的处理流程如下：

- 1) 实例 instance 转发包含目标主机 MAC 地址的数据包到 LinuxBridge 网桥 qbr。
- 2) LinuxBridge 网桥 qbr 对数据包采用安全组规则进行过滤处理。
- 3) LinuxBridge 网桥 qbr 将安全规则过滤后的数据转发到 OpenvSwitch 集成网桥 br-int。
- 4) OpenvSwitch 集成网桥 br-int 为数据包添加 Project 网络的内部 tag。
- 5) 如果 Project 网络是 GRE/VxLAN 类型，则转到步骤 6。

- ①如果 Project 网络是 VLAN 类型，则 OpenvSwitch 集成网桥 br-int 将数据包转发到 OpenvSwitchVLAN 网桥 br-vlan。
 - ② OpenvSwitchVLAN 网桥 br-vlan 使用 Project 网络的实际 VALN ID 替换掉 br-int 添加的 Project 网络内部 tag。
 - ③ OpenvSwitchVLAN 网桥 br-vlan 通过 VALN 接口将数据转发到网络节点。
- 6) 如果 Project 网络是 GRE/VxLAN 类型，则直接进入此步骤。
- ① OpenvSwitch 集成网桥 br-int 将数据包转发到 OpenvSwitch 隧道网桥 br-tun。
 - ② OpenvSwitch 隧道网桥 br-tun 将数据封装到 VxLAN/GRE 隧道中，并为封装后的数据包添加用以识别不同 Project 网络的 tag。
 - ③ OpenvSwitch 隧道网桥 br-tun 通过隧道网络接口将数据转发到网络节点。

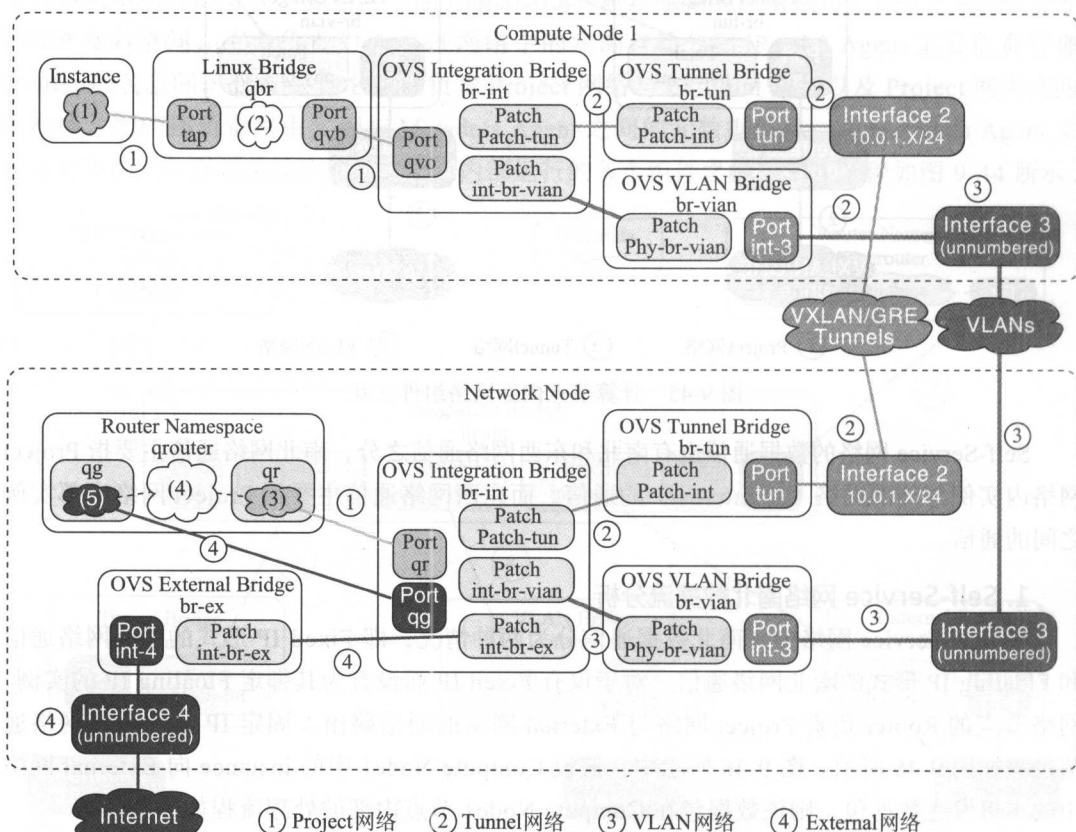


图 9-46 Fixed IP 实例南北网络通信

数据包到达网络节点后，在网络节点内部的处理流程如下：

- 1) 如果 Project 网络是 GRE/VxLAN 类型，则转到步骤 2。

- ①网络节点的 VLAN 网络接口将数据包转发到 OpenvSwitchVLAN 网桥 br-vlan。

② OpenvSwitch VLAN 网桥 br-vlan 将数据包转发到 OpenvSwitch 集成网桥 br-int。

③ OpenvSwitch 集成网桥 br-int 使用 Project 网络的内部 tag 替换掉 Project 的实际 VLAN ID。

2) 如果 Project 网络是 GRE/VxLAN 类型, 则直接进入此步骤。

① 网络节点隧道网络接口将数据转发到 OpenvSwitch 隧道网桥 br-tun。

② OpenvSwitch 隧道网桥 br-tun 解封经过 GRE/VxLAN 协议封装的数据包, 并为解封后的数据添加 Project 网络内部 tag。

③ OpenvSwitch 隧道网桥 br-tun 将数据包转发到 OpenvSwitch 集成网桥 br-int。

3) OpenvSwitch 集成网桥 br-int 将数据转发到 qrouter 命名空间的 qr 接口, qr 接口事先配置了 Project 网络的网关地址 (配置命令为: `neutron router-interface-add router_name subnet_net`)。

4) qrouter 命名空间中的 iptables 服务使用 qg 接口上的 IP 作为 Source IP 对数据包进行 SNAT 操作, qg 接口事先配置了 External 网络的网关地址 (配置命令: `neutron router-gateway-set router_name ext_net`)。

5) Router 通过 qg 接口转发数据包到 OpenvSwitch 集成网桥 br-int。

6) OpenvSwitch 集成网桥 br-int 转发数据包到 OpenvSwitch 外部网桥 br-ex。

7) OpenvSwitch 外部网桥 br-int 通过网络节点上的外部网络接口将数据包转发到 External 网络。

在生产环境中, 为了外部网络可以直接访问 Project 网络中的实例, 通常需要给实例指定 Floating IP, 并且 Floating IP 与实例的 Fixed IP 是一一对应的关系。具有 Floating IP 的实例南北网络通信过程如图 9-47 所示。假设位于 Compute Node1 中的实例 instance 接收来自位于 External 网络主机发送的数据包, 则数据包首先到达网络节点, 之后再进入计算节点。外网访问数据在网络节点中的处理过程如下:

1) 网络节点外网接口转发数据包到 OpenvSwitch 外部网桥 br-ex。

2) OpenvSwitch 外部网桥 br-ex 转发数据包到 OpenvSwitch 集成网桥 br-int。

3) OpenvSwitch 外部网桥 br-ex 转发数据包到 qrouter 命名空间的 qg 接口, qg 接口已事先配置了实例 Floating IP (配置命令: `neutron floatingip-create ext-net;nova floating-ip-associate instance_name floating_ip`)。

4) qrouter 命名空间中 iptables 服务使用 qr 接口的 IP 对数据包进行 DNAT 操作, qr 接口已事先配置了外网的网关地址 ((配置命令: `neutron router-gateway-set router_name ext_net`)。

5) qrouter 命名空间将数据包转发到 OpenvSwitch 集成网桥 br-int。

6) OpenvSwitch 集成网桥 br-int 为数据包添加 Project 网络内部 tag。

7) 如果 Project 网络是 GRE/VxLAN 类型, 则转到步骤 8。如果是 VLAN 类型, 则进入此步骤。

① OpenvSwitch 集成网桥 br-int 将数据包转发到 OpenvSwitch VLAN 网桥 br-vlan。

- ② OpenvSwitchVLAN 网桥 br-vlan 使用 Project 网络实际 VLAN ID 替换掉 Project 网络内部 tag。
- ③ OpenvSwitchVLAN 网桥 br-vlan 通过网络节点 VLAN 接口将数据转发到计算节点。
- 8) 如果 Project 网络是 GRE/VxLAN 类型, 则进入此步骤。
- ① OpenvSwitch 集成网桥 br-int 将数据包转发到 OpenvSwitch 隧道网桥 br-tun。
- ② OpenvSwitch 隧道网桥 br-tun 将数据封装到 VxLAN/GRE 隧道中, 并为封装后的数据包添加用以识别不同 Project 网络的隧道标记 tag。
- ③ OpenvSwitch 隧道网桥 br-tun 将数据包通过网络节点上的隧道网络接口转发到计算节点。

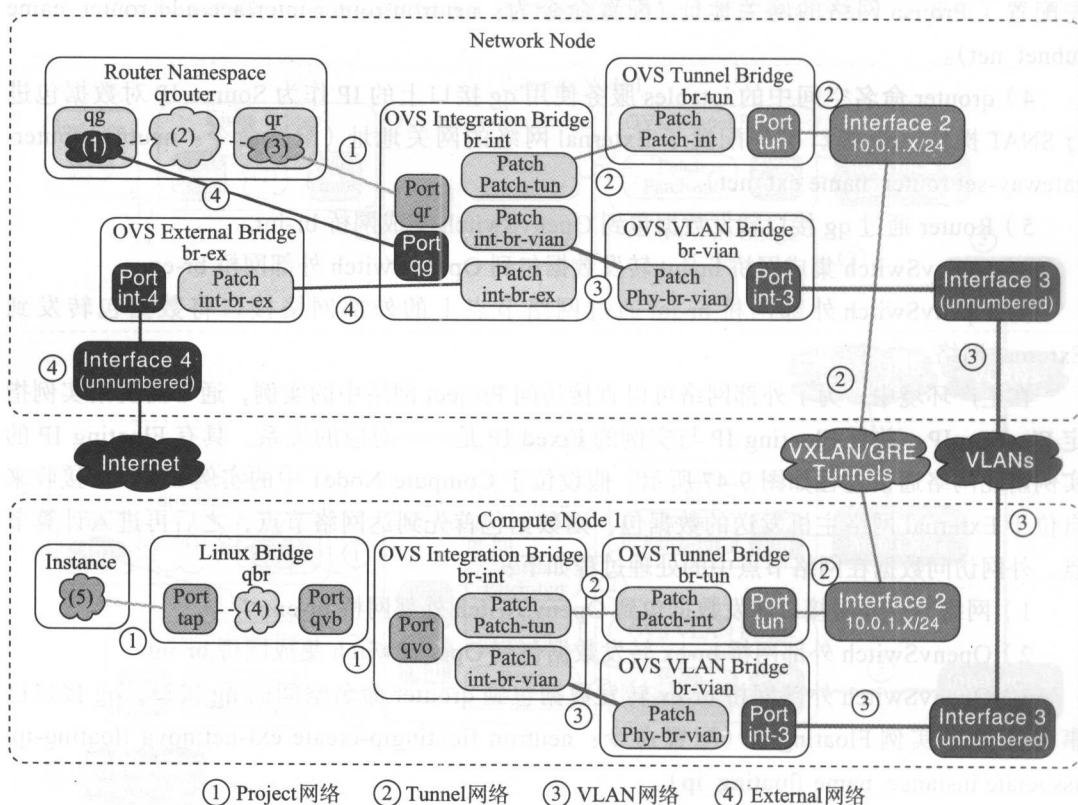


图 9-47 Floating IP 实例南北网络通信

数据进入计算节点后, 在计算节点内部的处理过程如下:

- 1) 如果 Project 网络是 VLAN 类型, 直接进入此步骤。如果是 RE/VxLAN 网络则转到步骤 2。

- ① 计算节点的 VLAN 接口将数据包转发到 OpenvSwitchVLAN 网桥 br-vlan。
- ② OpenvSwitchVLAN 网桥 br-vlan 将数据包转发到 OpenvSwitch 集成网桥 br-int。

- ③ OpenvSwitch 集成网桥 br-int 使用 Project 网络的内部 tag 替换掉实际 VLAN ID。
- 2) 如果 Project 网络是 GRE/VxLAN 类型, 则直接进入此步骤。
 - ① 计算节点上的隧道接口将数据转发到 OpenvSwitch 隧道网桥 br-tun。
 - ② OpenvSwitch 隧道网桥 br-tun 解封经过 GRE/VxLAN 协议封装的数据包, 并为解封后的数据添加 Project 网络内部 tag。
 - ③ OpenvSwitch 隧道网桥 br-tun 将数据包转发到 OpenvSwitch 集成网桥 br-int。
- 3) OpenvSwitch 集成网桥 br-int 转发数据到 LinuxBridge 网桥 qbr。
- 4) LinuxBridge 网桥 qbr 根据安全组规则过滤数据包。
- 5) LinuxBridge 网桥 qbr 将数据转发到实例 instance 的 tag 接口。

2. Self-Service 网络东西数据流分析

Project 网络内部的实例之间的通信称为东西网络通信, 东西网络通信通常分为两种, 即相同 Project 网络内部的实例通信和不同 Project 网络中的实例通信。此外, 对于东西网络通信, 实例是否绑定 Floating IP 并不影响通信方式。对于不同 Project 网络中的实例通信, 需要使用网络节点的路由功能对不同的 Project 网络进行路由, 不同 Project 网络中的实例东西数据通信过程如图 9-48 所示。假设位于 Compute Node1 且 Project 网络为 Network1 的实例 instance1 要与位于 Compute Node1 且 Project 网络为 Network2 的实例 instance2 通信, 则 instance1 发出的数据在 Compute Node1 上的处理过程如下:

- 1) instance1 的 tap 接口将包含目标 MAC 地址的数据转发到 LinuxBridge 网桥 qbr 中。
- 2) LinuxBridge 网桥 qbr 按照安全组规则对数据进行过滤处理。
- 3) LinuxBridge 网桥 qbr 中将数据转发到 OpenvSwitch 集成网桥 br-int。
- 4) OpenvSwitch 集成网桥 br-int 为数据包添加 Project Network1 的内部 tag。
- 5) 如果 Project 网络是 VLAN 类型, 则进入此步骤, 否则转到步骤 6。
 - ① OpenvSwitch 集成网桥 br-int 将数据转发到 OpenvSwitch VLAN 网桥 br-vlan。
 - ② OpenvSwitch VLAN 网桥 br-vlan 使用 Project Network1 的实际 VLAN ID 替换掉内部 tag。
 - ③ OpenvSwitch VLAN 网桥 br-vlan 通过 Compute Node1 的 VLAN 网络接口将数据转发到网络节点。
- 6) 如果 Project 网络为 GRE/VxLAN 类型, 则直接接入此步骤。
 - ① OpenvSwitch 集成网桥 br-int 将数据转发到 OpenvSwitch 隧道网桥 br-tun。
 - ② OpenvSwitch 隧道网桥 br-tun 将数据封装到 VxLAN/GRE 隧道中, 并为封装后的数据包添加用以识别不同 Project 网络的隧道标记 tag。
 - ③ OpenvSwitch 隧道网桥 br-tun 通过 Compute Node1 的隧道网络接口将数据转发到网络节点。

数据进入网络节点之后, 在网络节点中的处理过程如下:

- 1) 如果 Project 网络是 VLAN 类型, 则进入此步骤, 否则转到步骤 2。

- ①网络节点的 VLAN 网络接口将数据包转发到 OpenvSwitchVLAN 网桥 br-vlan。
 - ② OpenvSwitchVLAN 网桥 br-vlan 将数据包转发到 OpenvSwitch 集成网桥 br-int。
 - ③ OpenvSwitch 集成网桥 br-int 使用 Project 网络的内部 tag 替换掉 Project 的实际 VLAN ID。
- 2) 如果 Project 网络是 GRE/VxLAN 类型, 则直接进入此步骤。
- ①网络节点隧道网络接口将数据转发到 OpenvSwitch 隧道网桥 br-tun。
 - ② OpenvSwitch 隧道网桥 br-tun 解封经过 GRE/VxLAN 协议封装的数据包, 并为解封后的数据添加 Project 网络内部 tag。
 - ③ OpenvSwitch 隧道网桥 br-tun 将数据包转发到 OpenvSwitch 集成网桥 br-int。
- 3) OpenvSwitch 集成网桥 br-int 将数据转发到 qrouter 命名空间中的 qr-1 接口。qr-1 接口已配置有 Project Network1 的网关 IP (配置命令: `neutron router-interface-add router_name subnet1_name`)。
- 4) qrouter 命名空间将数据包路由到 qr-2 接口, qr-2 接口已配置有 Project Network2 的网关 IP (配置命令: `neutron router-interface-add router_name subnet2_name`)。
- 5) qrouter 命名空间将数据包转发到 OpenvSwitch 集成网桥 br-int。
- 6) OpenvSwitch 集成网桥 br-int 为数据包添加 Project Network2 的内部 tag。
- 7) 如果 Project 网络是 GRE/VxLAN 类型, 则转到步骤 8。如果是 VLAN 类型, 则进入此步骤。
- ① OpenvSwitch 集成网桥 br-int 将数据包转发到 OpenvSwitchVLAN 网桥 br-vlan。
 - ② OpenvSwitchVLAN 网桥 br-vlan 使用 Project 网络实际 VLAN ID 替换掉 Project 网络内部 tag。
 - ③ OpenvSwitchVLAN 网桥 br-vlan 通过网络节点 VLAN 接口将数据转发到 Compute Node2 计算节点。
- 8) 如果 Project 网络是 GRE/VxLAN 类型, 则进入此步骤。
- ① OpenvSwitch 集成网桥 br-int 将数据包转发到 OpenvSwitch 隧道网桥 br-tun。
 - ② OpenvSwitch 隧道网桥 br-tun 将数据封装到 VxLAN/GRE 隧道中, 并为封装后的数据包添加用以识别不同 Project 网络的隧道标记 tag。
 - ③ OpenvSwitch 隧道网桥 br-tun 将数据包通过网络节点上的隧道网络接口转发到 Compute Node2 计算节点。
- 数据进入 Compute Node2 计算节点后, 在 Compute Node2 中的处理过程如下:
- 1) 如果 Project 网络是 VLAN 类型, 则直接进入此步骤, 如果是 GRE/VxLAN 网络, 则转到步骤 2。
- ①计算节点的 VLAN 接口将数据包转发到 OpenvSwitchVLAN 网桥 br-vlan。
 - ② OpenvSwitchVLAN 网桥 br-vlan 将数据包转发到 OpenvSwitch 集成网桥 br-int。
 - ③ OpenvSwitch 集成网桥 br-int 使用 Project Network2 的内部 tag 替换掉实际 VLAN ID。

2) 如果 Project 网络是 GRE/VxLAN 类型, 则直接进入此步骤。

① 计算节点上的隧道接口将数据转发到 OpenvSwitch 隧道网桥 br-tun。

② OpenvSwitch 隧道网桥 br-tun 解封经过 GRE/VxLAN 协议封装的数据包, 并为解封后的数据添加 Project Network2 络内部 tag。

③ OpenvSwitch 隧道网桥 br-tun 将数据包转发到 OpenvSwitch 集成网桥 br-int。

3) OpenvSwitch 集成网桥 br-int 转发数据到 LinuxBridge 网桥 qbr。

4) LinuxBridge 网桥 qbr 根据安全组规则过滤数据包。

5) LinuxBridge 网桥 qbr 将数据转发到 Compute Node2 计算节点实例 instance2 的 tag 接口。

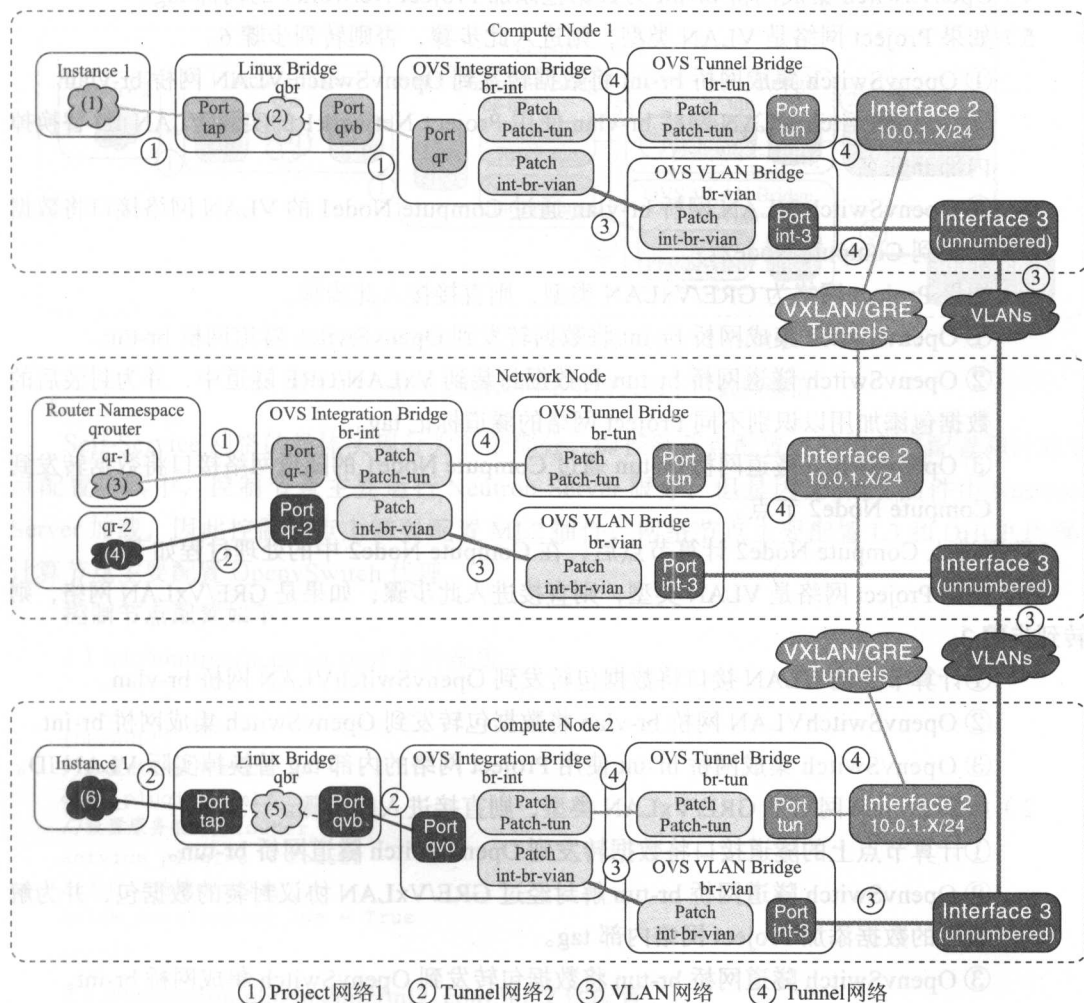


图 9-48 不同 Project 网络中的实例东西数据通信

相比不同 Project 网络中的实例通信, 相同 Project 网络内部实例通信要简单很多, 因为

相同 Project 网络内部实例通信无须进行路由，而是完全由 OpenvSwitch Agent 进行数据交换处理，因此也不需要网络节点参与。相同 Project 网络中的实例东西数据过程如图 9-49 所示。假设 Compute Node1 中的实例 instance1 要与 Compute Node2 上的实例 instance2 通信，并且两个实例同处相同的 Project 网络中，则 instance1 发出的数据包在 Compute Node1 上的处理过程如下：

- 1) instance1 的 tap 接口将包含目标 MAC 地址的数据转发到 LinuxBridge 网桥 qbr 中。
- 2) LinuxBridge 网桥 qbr 按照安全组规则对数据进行过滤处理。
- 3) LinuxBridge 网桥 qbr 中将数据转发到 OpenvSwitch 集成网桥 br-int。
- 4) OpenvSwitch 集成网桥 br-int 为数据包添加 Project Network1 的内部 tag。
- 5) 如果 Project 网络是 VLAN 类型，则进入此步骤，否则转到步骤 6。
 - ① OpenvSwitch 集成网桥 br-int 将数据转发到 OpenvSwitch VLAN 网桥 br-vlan。
 - ② OpenvSwitch VLAN 网桥 br-vlan 使用 Project Network1 的实际 VLAN ID 替换掉内部 tag。
 - ③ OpenvSwitch VLAN 网桥 br-vlan 通过 Compute Node1 的 VLAN 网络接口将数据转发到 Compute Node2。
- 6) 如果 Project 网络为 GRE/VxLAN 类型，则直接接入此步骤。
 - ① OpenvSwitch 集成网桥 br-int 将数据转发到 OpenvSwitch 隧道网桥 br-tun。
 - ② OpenvSwitch 隧道网桥 br-tun 将数据封装到 VxLAN/GRE 隧道中，并为封装后的数据包添加用以识别不同 Project 网络的隧道标记 tag。
 - ③ OpenvSwitch 隧道网桥 br-tun 通过 Compute Node1 的隧道网络接口将数据转发到 Compute Node2 节点。

数据进入 Compute Node2 计算节点后，在 Compute Node2 中的处理过程如下：

- 1) 如果 Project 网络是 VLAN 类型，则直接进入此步骤；如果是 GRE/VxLAN 网络，则转到步骤 2。
 - ① 计算节点的 VLAN 接口将数据包转发到 OpenvSwitch VLAN 网桥 br-vlan。
 - ② OpenvSwitch VLAN 网桥 br-vlan 将数据包转发到 OpenvSwitch 集成网桥 br-int。
 - ③ OpenvSwitch 集成网桥 br-int 使用 Project 网络的内部 tag 替换掉实际 VLAN ID。
- 2) 如果 Project 网络是 GRE/VxLAN 类型，则直接进入此步骤。
 - ① 计算节点上的隧道接口将数据转发到 OpenvSwitch 隧道网桥 br-tun。
 - ② OpenvSwitch 隧道网桥 br-tun 解封经过 GRE/VxLAN 协议封装的数据包，并为解封后的数据添加 Project 网络内部 tag。
 - ③ OpenvSwitch 隧道网桥 br-tun 将数据包转发到 OpenvSwitch 集成网桥 br-int。
- 3) OpenvSwitch 集成网桥 br-int 转发数据到 LinuxBridge 网桥 qbr。
- 4) LinuxBridge 网桥 qbr 根据安全组规则过滤数据包。
- 5) LinuxBridge 网桥 qbr 将数据转发到 Compute Node2 计算节点实例 instance2 的 tag 接口。

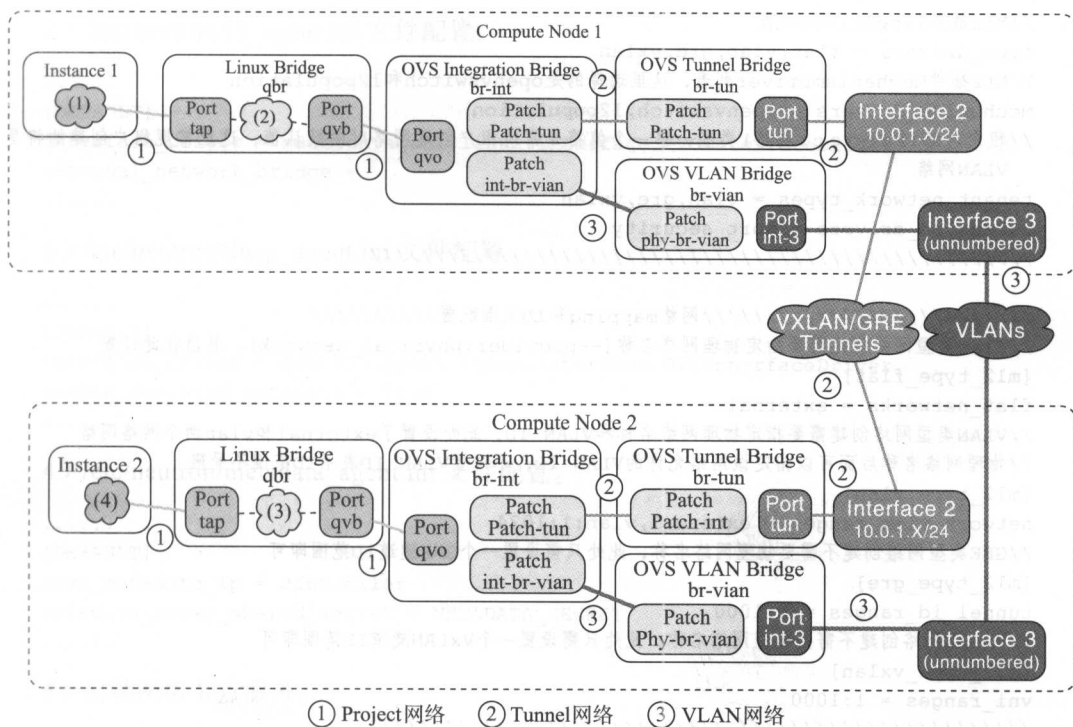


图 9-49 相同 Project 网络中的实例东西数据通信

Self-Service 网络的具体实现分为三个部分，即控制节点配置、网络节点配置和计算节点配置。其中，控制节点主要运行 Neutron-Server 服务，但是因为 ML2 插件由 Neutron-Server 加载，因此控制节点上需要配置 ML2 插件；网络节点主要配置 L3 和 DHCP 代理；计算节点主要配置 OpenvSwitch 代理。

控制节点配置如下：

1) /etc/neutron/neutron.conf 文件配置。

```
.....
[DEFAULT]
//设置核心插件为ML2
core_plugin = ml2
//设置服务插件为router
service_plugins = router
//允许IP地址overlay
allow_overlapping_ips = True
.....
```

2) /etc/neutron/plugins/ml2/ml2_conf.ini 文件配置。

```
.....
//////////ML2驱动和租户网络类型配置//////////
[ml2]
```



```
//ML2插件TypeDriver列表
type_drivers = flat,vlan,gre,vxlan
//ML2插件MechanismDriver列表, 这里选择的是openvswitch和l2population
mechanism_drivers = openvswitch,l2population
//租户网络(Project网络)类型, 第一个值将是常规租户创建网络时的默认值, 这里常规租户创建的将是
VLAN网络
tenant_network_types = vlan,gre,vxlan
extension_drivers = port_security
////////////////////////////////////
////////////////////////////////////网络mapping和ID范围配置////////////////////////////////////
//Flat类型网络创建需要指定物理网络名称(--provider:physical_network), 其值在此设置
[ml2_type_flat]
flat_networks = external
//VLAN类型网络创建需要指定物理网络名称和VLAN ID, 此处设置了external和vlan两个网络网络
//物理网络名称后面可以指定该网络允许的VLAN ID, 不设置VLAN ID表示ID范围不受限
[ml2_type_vlan]
network_vlan_ranges = external,vlan:1:1024
//GRE类型网络创建不需要物理网络名称, 此处只需设置一个GRE隧道ID范围即可
[ml2_type_gre]
tunnel_id_ranges = 1:1000
//VxLAN网络创建不需要物理网络名称, 此处只需设置一个VxLAN隧道ID范围即可
[ml2_type_vxlan]
vni_ranges = 1:1000
////////////////////////////////////
[securitygroup]
firewall_driver = iptables_hybrid
.....
```

3) 启动 Neutron-Server 服务。

```
systemctl start neutron-server.service
systemctl enable neutron-server.service
```

网络节点配置如下:

1) /etc/neutron/plugins/ml2/openvswitch_agent.ini 文件配置。

```
.....
[ovs]
//指定用于GRE/VxLAN的隧道IP接口地址(使用具体IP地址替换TUNNEL_INTERFACE_IP_ADDRESS)
local_ip = TUNNEL_INTERFACE_IP_ADDRESS
//设置对应物理网络的网桥, 此处的vlan和external两个物理网络名称必须与ml2_conf.ini文件中的设
定值一致
bridge_mappings = vlan:br-vlan,external:br-ex
[agent]
tunnel_types = gre,vxlan
//l2_population仅对GRE/VxLAN网络有效
l2_population = True
[securitygroup]
firewall_driver = iptables_hybrid
.....
```

2) /etc/neutron/l3_agent.ini 文件配置。

```
.....
[DEFAULT]
interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver
external_network_bridge =
.....
```

3) /etc/neutron/dhcp_agent.ini 文件配置。

```
.....
[DEFAULT]
interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver
enable_isolated_metadata = True
.....
```

4) /etc/neutron/metadata_agent.ini 文件配置。

```
.....
[DEFAULT]
nova_metadata_ip = controller
metadata_proxy_shared_secret = METADATA_SECRET
.....
```

5) 启动网络节点服务。

```
systemctl start openvswitch.service
systemctl enable openvswitch.service
systemctl start neutron-l3-agent.service
systemctl enable neutron-l3-agent.service
systemctl start dhcp-l3-agent.service
systemctl enable dhcp-l3-agent.service
systemctl start openvswitch-l3-agent.service
systemctl enable openvswitch-l3-agent.service
systemctl start metadata-l3-agent.service
systemctl enable metadata-l3-agent.service
```

计算节点配置如下：

1) /etc/neutron/plugins/ml2/openvswitch_agent.ini 文件配置。

```
[ovs]
local_ip = TUNNEL_INTERFACE_IP_ADDRESS
bridge_mappings = vlan:br-vlan
[agent]
tunnel_types = gre,vxlan
l2_population = True
[securitygroup]
firewall_driver = iptables_hybrid
```

2) 启动计算节点网络服务。

```
systemctl start openvswitch.service
systemctl enable openvswitch.service
```

```
systemctl start neutron-openvswitch-agent.service
systemctl enable neutron-openvswitch-agent.service
```

Neutron 服务在各个节点配置和启动完成后，即可进行 Project 网络的创建和验证。首先在控制节点验证各个节点上的网络服务是否正常启动。正常情况下应该显示如下：

```
#neutron agent-list
+-----+-----+-----+-----+-----+
+-----+
|      id      | agent_type          | host      | alive | admin_state_up|
+-----+-----+-----+-----+-----+
binary          |                      |           |       |                |
+-----+-----+-----+-----+-----+
|...8a64e6bea| Open vSwitch agent | compute1 | :- )  | True           |
neutron-openvswitch-agent|
|...a2c0d7529| Metadata agent     | network1 | :- )  | True           |
neutron-metadata-agent  |
|...6d4cda526| Open vSwitch agent | network1 | :- )  | True           |
neutron-openvswitch-agent|
|...b53e5b275| Open vSwitch agent | compute2 | :- )  | True           |
neutron-openvswitch-agent|
|...d12d084b2| DHCP agent         | network1 | :- )  | True           |
neutron-dhcp-agent      |
|...ea70591aa| L3 agent           | network1 | :- )  | True           |
neutron-l3-agent        |
+-----+-----+-----+-----+-----+
+-----+
```

创建一个 Flat 类型的 External 网络 ext-net:

```
[root@controller1 ~]#neutron net-create --router:external True --provider:physical_
network external --provider:network_type flat ext-net
```

为外部网络 ext-net 创建子网 ext-subnet:

```
[root@controller1 ~]# neutron subnet-create ext-net --name ext-subnet --allocation-
pool start=192.168.115.200,end=192.168.115.250 --disable-dhcp --gateway 192.
168.115.254 192.168.115.0/24
```

控制节点配置文件 ml2_conf.ini 中配置的租户网络类型第一个值为 vlan，因此普通租户创建的 Project 网络将默认是 VLAN 类型，仅有授权管理员用户才可创建 GRE 和 VxLAN 的 Project 网络。这里以管理员身份创建 GRE 类型的 project 网络，具体如下：

```
[root@controller1]# neutron net-create admin-net --provider:network_type gre
```

为 Project 网络 admin-net 创建子网 admin-subnet，子网 IP 可以是任意有效 IP：

```
[root@controller1]# neutron subnet-create admin-net --name admin-subnet --gateway
192.128.1.1 192.128.1.0/24
```

为租户创建 Router:

```
[root@controller1 neutron]# neutron router-create admin-router
```

将租户子网 admin-subnet 添加为 admin-router 的接口：

```
[root@controller1 ~]# neutron router-interface-add admin-router admin-subnet
Added interface ale42d12-e75a-41d1-9dcd-59ff2a1233e3 to router admin-router.
```

将外网 ext-net 的网关设置为 admin-router 的外网网关：

```
[root@controller1 ~]# neutron router-gateway-set admin-router ext-net
Set gateway for router admin-router
```

在网络节点上验证 qrouter 和 qdhcp 命名空间 (qdhcp 命名空间在创建实例后才会出现)：

```
[root@network1 ~]# ip netns
qrouter-f9848007-e2d5-4942-a611-a7587abdd0f8
qdhcp-961e20c8-943e-4580-ab46-b690bee35195
```

检查路由上是否出现两个端口，并且端口地址是否是 Project 网络和 External 网络的网关地址，对应 External 网络的地址应该是其子网网段的最低地址：

```
[root@controller1 ~]# neutron router-port-list admin-router
```

id	name	mac_address	fixed_ips
...05ff88a02a48		fa:16:3e:16:43:0a	{ "subnet_id": ..., "ip_address": "192.168.115.200" }
...59ff2a1233e3		fa:16:3e:dd:9e:05	{ "subnet_id": "8f9ca26b-bbae-43f3-87bd-e5952315226e", "ip_address": "192.128.1.1" }

在任一外网主机上 ping External 网络的网关地址：

```
[root@controller1 ~]# ping 192.168.115.200
PING 192.168.115.200 (192.168.115.200) 56(84) bytes of data:
64 bytes from 192.168.115.200: icmp_seq=1 ttl=64 time=24.3 ms
64 bytes from 192.168.115.200: icmp_seq=2 ttl=64 time=0.530 ms
64 bytes from 192.168.115.200: icmp_seq=3 ttl=64 time=1.40 ms
64 bytes from 192.168.115.200: icmp_seq=4 ttl=64 time=0.303 ms
```

Launch 一个实例，指定实例网络为 Project 网络 admin-subnet：

```
[root@controller1 ~]# nova boot --flavor 1 --image cirros-0.3.4-x86_64 --nic\
net-id=961e20c8-943e-4580-ab46-b690bee35195 --security-group default --key-
name admin-key instance1
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State	Networks
...216a9	instance1	ACTIVE	-	Running	admin-net=192.128.1.5

创建 Floating IP，并为实例绑定此 Floating IP：

```
[root@controller1 ~]# neutron floatingip-create ext-net
Created a new floatingip:
+-----+-----+
| Field | Value |
+-----+-----+
.....
| floating_ip_address | 192.168.115.201 |
.....
[root@controller1 ~]# nova floating-ip-associate instance1 192.168.115.201
[root@controller1 ~]# nova list
+-----+-----+-----+-----+-----+-----+
| ID | Name | Status | Task State | Power State | Networks |
+-----+-----+-----+-----+-----+-----+
| ...3216a9 | instance1 | ACTIVE | - | Running | admin-net=192.128.1.5, 192.168.115.201 |
+-----+-----+-----+-----+-----+-----+
```

在控制节点或任一外网主机上 ping 实例 Floating IP:

```
[root@controller1 ~]# ping 192.168.115.201
PING 192.168.115.201 (192.168.115.201) 56(84) bytes of data.
64 bytes from 192.168.115.201: icmp_seq=1 ttl=62 time=1.41 ms
64 bytes from 192.168.115.201: icmp_seq=2 ttl=62 time=3.01 ms
64 bytes from 192.168.115.201: icmp_seq=3 ttl=62 time=0.898 ms
64 bytes from 192.168.115.201: icmp_seq=4 ttl=62 time=0.674 ms
```

到此，Self-Service 网络已经成功创建并可以提供正常网络服务。在 Self-Service 网络中，只要管理员创建好 Provider 物理网络后，租户便可以按需创建和删除符合自己需求的 Project 网络。

9.5.2 Self-Service 网络高可用

在 Juno 版本发行前，由于 Neutron 的 L3 Service 全部集中在网络节点，Self-Service 网络架构中的网络节点负载了几乎全部 OpenStack 集群实例的南北网络数据流和东西网络数据流，从而造成了网络节点的瓶颈和单点故障，因此 Self-Service 网络架构在性能和可用性上都存在很大的缺陷和隐患。因而早期基于 OpenStack 的云计算环境为了提供高可用和高性能的网络，通常采用类似 Provider 网络架构这种借助物理网络设备来实现的虚拟化网络方案，但是，Provider 网络过多引入物理网络设备来实现云计算网络功能，并不能真正体现软件定义网络的思想。过多的硬件实现致使 OpenStack 网络资源极不灵活，如普通租户不能自助创建和管理自己的网络资源、租户对网络资源的申请必须由云管理员来完成等等。

OpenStack 的 Neutron 团队一直致力于解决 Self-Service 网络架构的性能和高可用性问题。在 Juno 版本发行之前，Neutron 的高可用性便一直集中在 L3 Agent 的实现上，由于 L3 Service 是一种有状态的服务，因此常规的 HAProxy 负载均衡方案虽然可以解决 Neutron-Serve 这类无状态服务的高可用性问题，但不能解决 L3 的高可用性问题，因此社区针对 L3 Agent 基于 VRRP 协议开发了原生高可用方案（NativeHighAvailability）L3 HA。L3 HA

在很多厂商高可用解决方案中被使用,如 RedHat 采用的 Pacemaker 集群方案中,Neutron 资源代理脚本 NeutronScale 便是基于 L3 HA 的一种实现方案。然而,L3 HA 仍然存在一些问题,例如为了实现 L3 HA,必须提供两个以上的网络节点,而多余的网络节点仅处于 Standby 状态,真正工作的网络节点仍然仅有一个,因此 L3 HA 方案增加了额外的硬件开销。更重要的是,当集群计算节点数目不断扩展时,由于 L3 HA 方案仍然是单网络节点工作,因此网络节点的性能瓶颈仍然存在,可以说 L3 HA 方案以节点冗余的方式解决了网络节点单点故障问题,但是网络节点性能问题依然存在。

Juno 版本发行后,L3 Agent 的高可用方案中增加了分布式虚拟路由 DVR。DVR 的实现解决了 L3 Agent 的分布式实现问题,因此不仅网络节点的单点故障问题被解决,网络负载也由网络节点分散到计算节点,因此相对 L3 HA 方案,DVR 是一种更理想的网络高可用实现方案。但是,在 Mitaka 版本发行前,DVR 中的 L3 高可用并不彻底,L3 SNAT 功能仍然被遗留在网络节点上,或者说网络节点的 DNAT 南北网络通信和东西网络通信被分流到各个计算节点,但是 SNAT 功能仍然需要网络节点来实现,并且 Mitaka 之前的 Neutron 项目中,DVR 功能和 L3 HA 不能同时开启,这意味着网络节点上的 SNAT 依然是单点故障。不过,SNAT 功能保留的目的主要是为了节约宝贵的公网 IP,如果实例数目有限,公网 IP 地址丰富,则可以为每个实例绑定 Floating IP,从而直接停用 SNAT 功能。

到了 Mitaka 版本,Neutron 网络在高可用和性能瓶颈两个维度上终于被彻底解决,DVR 支持与 L3 HA 同时启用,在 DVR 分流网络负载至各个计算节点的同时,遗留在网络节点的 L3 SNAT 通过 L3 HA 解决了单点故障问题。因此,Mitaka 版本中的 Neutron 项目不仅彻底解决了 OpenStack 网络节点的单点故障问题,同时解决了网络节点的性能瓶颈问题。具体而言,Mitaka 版本后,用户创建的 L3 Router 可以同时具备分布属性和 HA 属性,具体如下:

```
[root@controller1 ~]# neutron router-list
```

id	name	external_gateway_info	distributed	ha
...d0f8	admin-router	{"network_id": "bde23e1c-1026-4639-ba31-6b3ee38bef00", "enable_snat": true, "external_fixed_ips": [{"subnet_id": "da5cece8-3454-487c-b213-ad7beb79f5c9", "ip_address": "192.168.115.200"}]}	True	True

就目前而言,用户在部署高可用网络并需要解决 L3 高可用问题时,可以选择传统的 L3 HA 方案或者 DVR 方案,同时还可以部署最新的 Mitaka 版本,从而采用 DVR 与 L3 HA 同时启用的方案。

9.6 L3 HA 高可用方案

9.6.1 L3 HA 高可用部署实现

L3 HA 是对 9.5 节传统 Self-Service 网络实现中 Layer-3 Service 的增强实现，与传统的 Self-Service 网络架构类似，特定租户的 Project 网络数据仍然只汇聚到一个网络节点上的活动路由中进行路由转发，而提供 HA 功能所增加的网络节点并不会负载此租户的网络流量。因此 L3 HA 需完成的首要任务是网络节点的单点故障问题，而不是集群网络的带宽限制和瓶颈问题。不过从整体上看，L3 HA 方案支持租户的多个路由或不同租户的路由随机分布到多个网络节点，这在一定程度上缓减了单个网络节点负载全部路由时造成的压力，因此也在一定程度上增加了网络节点的扩展性。L3 HA 高可用方案主要采用基于 VRRP 协议的 Keepalived 来实现 Self-Service 网络架构下 L3 Service 的快速故障转移。L3 HA 的设计思路可以概括如下。

- ❑ 在多个网络节点上部署 L3 Agent 服务，当租户创建高可用 L3 Router 时，在多个运行 L3 Agent 的网络节点上同步创建多个 L3 Router 实例，不同网络节点上的 L3 Router 完全相同。
- ❑ 借助基于 VRRP 协议的软件（如 Keepalived），使得多个网络节点上的 L3 Router 具有不同的运行状态，即其中仅有某个 L3 Router 是 Master 状态，而其他 L3 Router 都是 Standby 状态。正常情况下，由 Master 向虚机提供路由服务，当 Master 故障时，重新从 Standby 状态的 L3 Router 中选举新的 Master。目前 Neutron 中 L3 HA 使用 Keepalived 来实现此功能。
- ❑ L3 Router 故障切换时，Router 实例并不在网络节点之间迁移，而是将 Router 对外服务的 VIP 漂移到另外的网络节点从而实现路由访问的持续性。

从实现过程的本质上来看，Neutron 的 L3 HA 功能针对的是 qrouter 命名空间的高可用，在用户部署多个 L3 Agent 网络节点并启用 L3 HA 功能后，用户在控制节点创建的 Router 便具有 HA 功能。以部署了两个 L3 Agent 服务的网络节点为例，当用户创建具有 HA 功能的 Router 后，两个网络节点上 L3 Router 的高可用架构如图 9-50 所示。

图 9-50 所示中，每个 qrouter 命名空间中均有一个 QR 和 QG 接口，两个接口分别连接 Project 网络和 External 网络。QG 和 QR 接口是所有 qrouter 命名空间都存在的接口，并非仅有高可用 qrouter 命名空间才有，并且全部 qrouter 命名空间中的 QR 和 QG 具有完全相同的设备名和 MAC 地址，可以认为不同网络节点上的 qrouter 命名空间源自同一个副本。除此之外，命名空间中还有一个 HA 端口，这是 HA Router 特有的端口，此端口主要用于 VRRP 广播传递。每一组 HA Routers 中包含两个 VIP，一个是 Project 网络的网关 IP，另一个是 External 网络的网关 IP。这两个 VIP 并不在路由创建后立刻出现（HA Router 创建后 qrouter 命名空间仅有 HA 端口），而是在用户为路由设置 Project 网络接口和 External 网络网关时才会出现。此外，这两个 VIP 同时位于某个处于 Master 状态的 Router 上，当发生

Router 故障切换时，两个 VIP 同时切换到新的 Master Router 上。由于故障时两个 VIP 自动迁移不受影响，因此 Project 网络和 External 网络对 Router 的访问也不会受到影响。

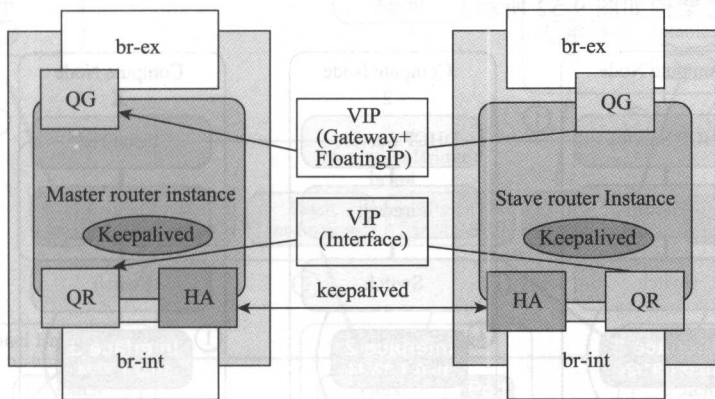


图 9-50 HA Router 内部结构

本节以实现 5 节点的 OpenStack 集群为例来讲解 L3 HA 高可用方案的实现并对其进行分析，其中两个为网络节点，用于实现 L3 HA，另外两个为计算节点，剩余一个为控制节点。在实现过程中，集群网络仍然采用基于 OpenvSwitch 实现的传统 Self-Service 架构，并且两个网络节点部署相同的服务并进行相同的配置，两个计算节点也部署相同服务并进行相同配置。控制节点、网络节点和计算节点所运行的网络服务如图 9-51 所示。

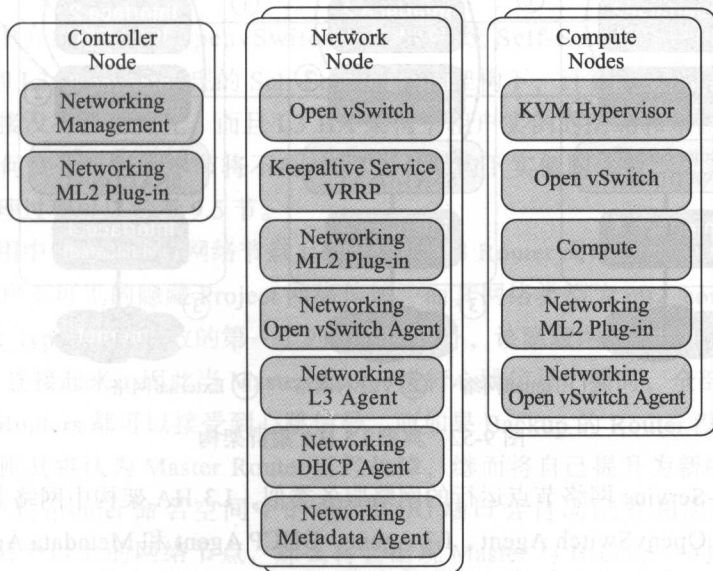


图 9-51 不同类型节点网络服务部署情况

从集群架构上看，L3 HA 实现的 Self-Service 网络架构与传统单网络节点 Self-Service

网络架构并无太大差异，主要区别在于传统 Self-Service 网络中单网络节点变为 L3 HA 架构中的多网络节点。多网络节点基于 Keepalived 实现 L3 路由的高可用。典型的 L3 HA Self-Service 网络架构如图 9-52 所示。

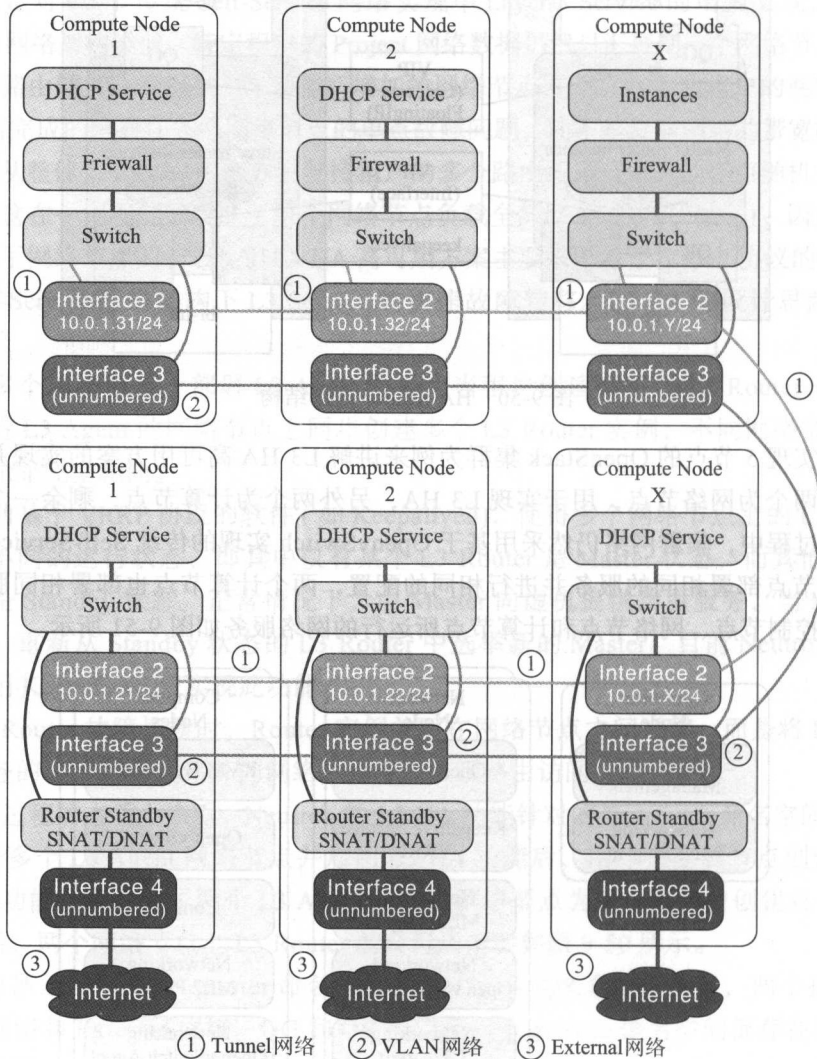


图 9-52 典型 L3 HA 集群架构

与传统 Self-Service 网络节点运行的网络服务类似，L3 HA 架构中网络节点主要运行的网络服务仍然是 OpenvSwitch Agent、L3 Agent、DHCP Agent 和 Metadata Agent，不同的地方在于 L3 Agent。在 L3 HA 架构的网络节点中，L3 Agent 不仅负载 qrouter 命名空间管理，还要负载基于 VRRP 协议的 Keepalived 高可用集群管理。因此，从网络节点内部 qrouter 命名空间来看，L3 HA 架构下的 qrouter 命名空间和集成网桥 br-int 将会多出一对 HA 端口，

L3 HA 架构下的网络节点内部组件连接如图 9-53 所示。

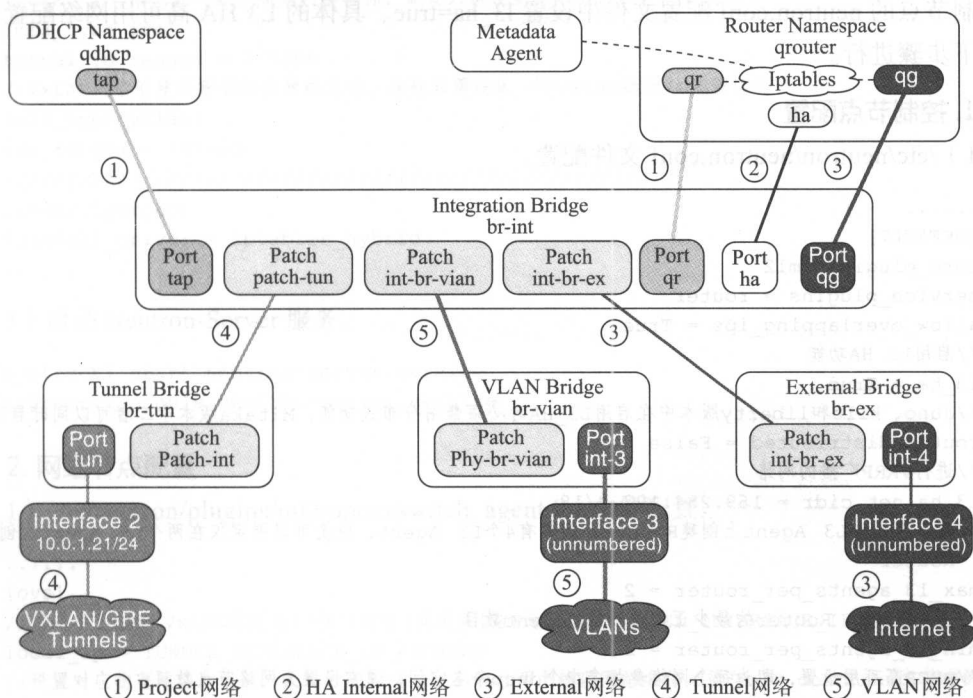


图 9-53 L3 HA 架构网络节点内部组件连接

与 9.5 节中介绍的基于 OpenvSwitch 实现的传统 Self-Service 网络架构相比，基于 OpenvSwitch 和 L3 HA 增强实现的 Self-Service 网络架构下，计算节点内部运行服务和组件之间的通信连接没有任何变化，而且 L3 HA 架构下租户实例的南北和东西网络数据流处理过程也没有任何变化，因此本节将不再对 L3 HA 架构下实例网络数据处理过程进行分析，具体的数据处理过程可以参考 9.5 节。

在实际应用中，位于某个网络节点上的 Master L3 Router 周期性发出心跳检测信号，心跳信号使用租户不可见的隐藏 Project 网络传输，而其网络类型为 ml2_conf.ini 配置文件中 tenant_network_types 配置参数的第一个列表值。此外，该隐藏网络将属于特定 Project 的全部 HA Routers 连接起来，因此当 Master Router 进行心跳信号广播时，全部接入该网络中正常运行的 HA Routers 都可以接受到心跳信号，而如果 Backup 的 Router 没有接受到 Master 广播的信号，则其将认为 Master Router 已经故障，继而将自己提升为新的 Master Router，在提升过程中其 qrouter 命名空间中的 QR 和 QG 端口会自动配置相应的 VIP。如果 L3 HA 环境中有两个以上的网络节点，那么将会出现 Master 与 Backup 一对多的关系，此时 Backup Routers 中优先级最高的将成为新的 Master Router。由于 Neutron 的 L3 HA 机制对全部 Routers 使用完全相同的优先级，因此 VRRP 协议在 Master 故障时将提升 IP 地址最大的 Backup Router 成为新的 Master Router。

L3 HA 网络架构的部署过程与传统 Self-Service 网络的实现类似，其不同之处只是需要在控制节点的 `neutron.conf` 配置文件中设置 `l3_ha=true`、具体的 L3 HA 高可用网络配置可以按如下步骤进行。

1. 控制节点配置

1) /etc/neutron/neutron.conf 文件配置。

```
.....
[DEFAULT]
core_plugin = ml2
service_plugins = router
allow_overlapping_ips = True
//启用L3 HA功能
l3_ha = True
//Juno、kilo和liberty版本中在启用L3_HA时必须禁用分布式功能，Mitaka版本中二者可以同时启用
router_distributed = False
//进行VRRP广播的网络
l3_ha_net_cidr = 169.254.192.0/18
//最多在几个L3 Agent上创建Router，例如有4个L3 Agent，但是可以限定仅在两个L3 Agent上创建HA
Router
max_l3_agents_per_router = 2
//可以创建HA Router的最少正常运行L3 Agent数目
min_l3_agents_per_router = 2
//DHCP高可用设置，即为每个网络参加多少个dhcp命名空间，通常设置为网络节点数目
dhcp_agents_per_network = 2
.....
```

2) /etc/neutron/plugins/ml2/ml2_conf.ini 文件配置。

```
.....
//////////ML2驱动和租户网络类型配置//////////
[ml2]
//ML2插件TypeDriver列表
type_drivers = flat,vlan,gre,vxlan
//ML2插件MechanismDriver列表，这里选择的是openvswitch和l2population，l2_population仅
//对类型为GRE/VxLAN的租户网络有效
mechanism_drivers = openvswitch,l2population
//租户网络(Project网络)类型，第一个值将是常规租户创建网络时的默认值，这里常规租户创建的将是VLAN
//网络，Master Router进行心跳信号传递的网络类型也默认是第一个值，即VLAN网络
tenant_network_types = vlan,gre,vxlan
extension_drivers = port_security
//////////
//////////网络mapping和ID范围配置//////////
//Flat类型网络创建需要指定物理网络名称(--provider:physical_network)，其值在此设置
[ml2_type_flat]
flat_networks = external
//VLAN类型网络创建需要指定物理网络名称和VLAN ID，此处设置了external和vlan两个网络网络
//物理网络名称后面可以指定该网络允许的VLAN ID，不设置VLAN ID表示ID范围不受限
[ml2_type_vlan]
```

```

network_vlan_ranges = external,vlan:1:1024
//GRE类型网络创建不需要物理网络名称，此处只需设置一个GRE隧道ID范围即可
[m12_type_gre]
tunnel_id_ranges = 1:1000
//VxLAN网络创建不需要物理网络名称，此处只需设置一个VxLAN隧道ID范围即可
[m12_type_vxlan]
vni_ranges = 1:1000
////////////////////////////////////
[securitygroup]
firewall_driver = iptables_hybrid
.....

```

3) 启动 Neutron-Server 服务。

```

systemctl start neutron-server.service
systemctl enable neutron-server.service

```

2. 网络节点配置

1) /etc/neutron/plugins/ml2/openvswitch_agent.ini 文件配置。

```

.....
[ovs]
//指定用于GRE/VxLAN的隧道IP接口地址(使用具体IP地址替换TUNNEL_INTERFACE_IP_ADDRESS)
local_ip = TUNNEL_INTERFACE_IP_ADDRESS
//设置对应物理网络的网桥，此处的vlan和external两个物理网络名称必须与ml2_conf.ini文件中的设定
//值一致
bridge_mappings = vlan:br-vlan,external:br-ex
[agent]
tunnel_types = gre,vxlan
//l2_population仅对GRE/VxLAN网络有效
l2_population = True
[securitygroup]
firewall_driver = iptables_hybrid
.....

```

2) /etc/neutron/l3_agent.ini 文件配置。

```

.....
[DEFAULT]
interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver
external_network_bridge =
.....

```

3) /etc/neutron/dhcp_agent.ini 文件配置。

```

.....
[DEFAULT]
interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver
enable_isolated_metadata = True
.....

```

4) /etc/neutron/metadata_agent.ini 文件配置。

```
.....
[DEFAULT]
nova_metadata_ip = controller
metadata_proxy_shared_secret = METADATA_SECRET
.....
```

5) 启动网络节点服务。

```
systemctl start openvswitch.service
systemctl enable openvswitch.service
systemctl start neutron-l3-agent.service
systemctl enable neutron-l3-agent.service
systemctl start dhcp-l3-agent.service
systemctl enable dhcp-l3-agent.service
systemctl start openvswitch-l3-agent.service
systemctl enable openvswitch-l3-agent.service
systemctl start metadata-l3-agent.service
systemctl enable metadata-l3-agent.service
```

3. 计算节点配置

1) /etc/neutron/plugins/ml2/openvswitch_agent.ini 文件配置。

```
[ovs]
//使用具体的隧道IP地址替换TUNNEL_INTERFACE_IP_ADDRESS
local_ip = TUNNEL_INTERFACE_IP_ADDRESS
bridge_mappings = vlan:br-vlan
[agent]
tunnel_types = gre,vxlan
l2_population = True
[securitygroup]
firewall_driver = iptables_hybrid
```

2) 启动计算节点网络服务。

```
systemctl start openvswitch.service
systemctl enable openvswitch.service
systemctl start neutron-openvswitch-agent.service
systemctl enable neutron-openvswitch-agent.service
```

到此，基于 OpenvSwitch 和 L3 HA 增强实现的 Self-Service 网络已配置完成。在正式创建 Project 网络和 HA Router 之前先验证各个节点网络服务是否正常启动。正常情况下应该看到如下的输出：

```
[root@controller1 ~]# neutron agent-list
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| id | agent_type | host | availability_zone | alive | admin_state_up |
+-----+-----+-----+-----+-----+-----+
binary |
```

```

+-----+-----+-----+-----+-----+-----+
+-----+
|...526 | Open vSwitch agent| network1|          | :- ) | True      |
neutron-openvswitch-agent |          |
|...275 | Open vSwitch agent| network2|          | :- ) | True      |
neutron-openvswitch-agent |          |
|...4b2 | DHCP agent        | network2| nova     | :- ) | True      |
neutron-dhcp-agent        |          |
|...1aa | DHCP agent        | network1| nova     | :- ) | True      |
neutron-dhcp-agent        |          |
|...e5a | L3 agent           | network2| nova     | :- ) | True      |
neutron-l3-agent          |          |
|...666 | L3 agent           | network1| nova     | :- ) | True      |
neutron-l3-agent          |          |
|...0a9 | Metadata agent      | network1|          | :- ) | True      |
neutron-metadata-agent    |          |
|...f49 | Open vSwitch agent| compute2|          | :- ) | True      |
neutron-openvswitch-agent |          |
|...b7f | Open vSwitch agent| computel|          | :- ) | True      |
neutron-openvswitch-agent |          |
|...5a7 | Metadata agent      | network2|          | :- ) | True      |
neutron-metadata-agent    |          |
+-----+-----+-----+-----+-----+-----+
+-----+

```

9.6.2 L3 HA 高可用验证与分析

L3 HA 增强实现的 Self-Service 网络以增加网络节点的形式提高 Neutron 网络服务的高可用性。虽然 L3 HA 网络架构方案仍然采用网络汇聚的方式提供 L3 服务，因而无法从根本上解决网络带宽限制和瓶颈问题，但是 L3 HA 方案确实解决了网络节点的单点故障问题。根据 L3 HA 的设计，当用户在配置文件中启用 L3 HA 功能后，租户创建的路由被自动设置为 HA Router，而且不同租户 HA Router 的 Master 或者同一个租户不同 HA Router 的 Master 随机分布到不同的网络节点上。当任一网络节点故障时，位于其上的 MasterRouter 将自动转移到其他网络节点上，而转移过程本质上是 qrouter 命名空间中 VIP 的迁移。由于故障路由实例中对外服务的 VIP 自动迁移到正常节点上的 qrouter 命名空间，因此路由对外服务不受影响。

在基于前一节已经实现 L3 HA 网络部署的前提下，本节将对 L3 HA 的高可用功能进行探索和验证。由于 L3 HA 网络架构中租户实例的南北和东西网络数据处理过程与 9.5 节中传统的 Self-Service 网络架构类似，因此本节不再对租户实例数据流向进行分析。在验证 L3 Router 的高可用性之前，用户首先需要创建 Project 网络和 External 网络，具体的网络创建和 L3 HA 功能验证过程如下所述。

1. 网络创建

为了对 L3 HA Router 进行验证, 首先需要创建 Project 网络和 External 网络。这里以创建 Flat 类型的 External 网络和 GRE 类型的 Project 网络为例。当然用户也可以创建 Flat 类型的 External 网络和 VxLAN 类型的 Project 网络或者 VLAN 类型的 External 网络和 VLAN 类型的 Project 网络。Project 网络和 External 网络的创建过程如下。

1) 创建 Flat 类型 External 网络, 网络名为 ext-net。

```
[root@controller1 ~]# neutron net-create ext-net --router:external True \
--provider:physical_network external --provider:network_type flat
```

2) 创建 External 网络子网, 网络名为 ext-subnet, 网络可用 IP 范围为 192.168.115.200~192.168.115.250。

```
[root@controller1 ~]# neutron subnet-create ext-net 192.168.115.0/24 --name ext-
subnet --allocation-pool start=192.168.115.200,end=192.168.115.250 --disable-
dhcp --gateway 192.168.115.254
```

3) 创建 Project 网络, 网络名为 admin-net。根据控制节点 ml2_conf.ini 中 tenant_network_types 配置, 常规租户只能创建 VLAN 类型的租户网络, 这里加载管理员权限后可以创建 GRE 租户网络。

```
[root@controller1 ~]# neutron net-create admin-net --provider:network_type gre
```

4) 创建 Project 网络子网, 名称为 admin-subnet。租户子网采用 DHCP 自动分配 IP, 网络为 192.128.1.0/24, 网关为 192.128.1.1。

```
[root@controller1 ~]# neutron subnet-create admin-net 192.128.1.0/24 --name
admin-subnet --gateway 192.128.1.1
```

5) 验证创建的 Project 网络和 External 网络。

```
[root@controller1 ~]# neutron net-list
```

id	name	subnets
...98ba	admin-net	c0921fde-8c5b-4780-81a6-5da5e5b007d4 192.128.1.0/24
...464f	ext-net	8f04fc59-a372-41f7-be90-7e36a84b096e 192.168.115.0/24

```
[root@controller1 ~]# neutron subnet-list
```

id	name	cidr	allocation_pools
...7d4	admin-subnet	192.128.1.0/24	{"start": "192.128.1.2", "end": "192.128.1.254"}
...96e	ext-subnet	192.168.115.0/24	{"start": "192.168.115.200", "end": "192.168.115.250"}

2. HA Router 创建与高可用性验证

1) HA Router 创建, 这里创建一个名为 l3-router 的 HA Router, 创建完成后检查 Router 是否具有 ha=True 属性。

```
[root@controller1 ~]# neutron router-create l3-router
[root@controller1 ~]# neutron router-list
```

id	name	external_gateway_info	distributed	ha
...e86d	l3-router	null	False	True

2) 确定运行 Master Router 的网络节点, Neutron 提供了命令 l3-agent-list-hosting-router 来查看。

```
[root@controller1 ~]# neutron l3-agent-list-hosting-router l3-router
```

id	host	admin_state_up	alive	ha_state
...d099f3e5a	network2	True	:-)	standby
...c15814666	network1	True	:-)	active

3) 查看 Neutron 中网络变化。

```
[root@controller1 ~]# neutron net-list
```

id	name	subnets
...464f	ext-net	8f04fc59-a372-41f7-be90-7e36a84b096e 192.168.115.0/24
...98ba	admin-net	c0921fde-8c5b-4780-81a6-5da5e5b007d4 192.128.1.0/24
...85b8	HA network tenant ...f918a793	dea72fe1-a99c-4f85-8901-43b376ca4194 169.254.192.0/18

可以看到, 创建 HA Router 后, Neutron 中新增一个租户网络 169.254.192.0/18, 该网络以“HA network tenant tenant_ID”的形式命名。这是一个以当前租户身份信息自动创建的 HA 网络, 租户实例不能使用此网络, Keepalived 进程使用此网络进行心跳信号广播, 可以将此网络认为是不同网络节点路由实例之间的高可用网络。

4) 查看各个网络节点 qrouter 命名空间。

```
//网络节点1
```

```
[root@network1 ~]# ip netns
qrouter-fb6fe477-e382-42d8-b303-81ca8621e86d
qdhcp-09d6e2a3-1efe-4561-b6eb-ffe14a4d98ba
```

```
//网络节点2
```

```
[root@network2 ~]# ip netns
```

```
qrouter-fb6fe477-e382-42d8-b303-81ca8621e86d
qdhcp-09d6e2a3-1efe-4561-b6eb-ffe14a4d98ba
```

可以看到，network1 和 network2 节点中分别创建了一个 qrouter 和 qdhcp 命名空间（这里之所以以字母“q”命名，是因为 Neutron 项目的历史原因，Neutron 的前身叫 Quantum）。注意观察 network1 和 network2 节点的 qrouter 命名空间 ID 与控制节点的 l3-router 路由的 ID 是否一致，正常情况下完全一致。

5) 查看网络节点 qrouter 命名空间接口。

```
//网络节点1(qrouter命名空间lo端口已被省略)
[root@network1 ~]# ip netns exec qrouter-fb6fe477-e382-42d8-b303-81ca8621e86d ip
addr show
41: ha-140a8c78-33:<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue state
UNKNOWN
    link/ether fa:16:3e:b6:10:ee brd ff:ff:ff:ff:ff:ff
    inet 169.254.192.2/18 brd 169.254.255.255 scope global ha-140a8c78-33
        valid_lft forever preferred_lft forever
    inet 169.254.0.1/24 scope global ha-140a8c78-33
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:feb6:10ee/64 scope link
        valid_lft forever preferred_lft forever
//网络节点2(qrouter命名空间lo端口已被省略)
[root@network2 ~]# ip netns exec qrouter-fb6fe477-e382-42d8-b303-81ca8621e86d ip
addr show
40: ha-6c44b430-74:<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue state
UNKNOWN
    link/ether fa:16:3e:ac:f6:96 brd ff:ff:ff:ff:ff:ff
    inet 169.254.192.1/18 brd 169.254.255.255 scope global ha-6c44b430-74
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:feac:f696/64 scope link
        valid_lft forever preferred_lft forever
```

网络节点 network1 上的 qrouter 命名空间中有一个 HA 接口 ha-140a8c78-33，其 IP 为 169.254.192.2；网络节点 network2 上的 qrouter 命名空间中也有一个 HA 接口 ha-6c44b430-74，其 IP 为 169.254.192.1。命名空间 HA 接口是 Keepalived，其是用于 VRRP 协议广播心跳信号的端口。如下输出显示了 network1 节点上 Master Router 的广播信息：

```
[root@network1 ~]# ip netns exec qrouter-fb6fe477-e382-42d8-b303-81ca8621e86d
tcpdump -ni ha-140a8c78-33
listening on ha-140a8c78-33, link-type EN10MB (Ethernet), capture size 65535
bytes
16:01:03.775430 IP 169.254.192.2 > 224.0.0.18: VRRPv2, Advertisement, vrid 1,
prio 50, authtype none, intvl 2s, length 20
```

可以看到 HA 端口 ha-140a8c78-33 向广播地址 224.0.0.18 发送信息，广播地址是由 VRRP 协议决定的。上述输出信息中，vrid 表示虚拟路由 ID，租户的每个 HA Router 对应着一个唯一 ID。由于位数限制，每个租户最多可以创建 255 个 HA Router。prio 是网络节点上 qrouter 实例的优先级，目前每个网络节点上的路由实例具有完全相同的优先级，并且

用户目前不能自定义设置优先级。authtype 表示网络节点上路由实例之间通信的认证方式，默认无须认证。intvl 表示广播信号的发送间隔，默认为 2s，广播间隔是可以自定义设置的。

6) 为 HA Router 设置 Project 网络接口和 External 网络网关。

```
[root@controller1 ~]# neutron router-interface-add l3-router admin-subnet
Added interface 5eeca55-8f4c-4a9e-b58c-eae69346528d to router l3-router.
[root@controller1 ~]# neutron router-gateway-set l3-router ext-net
Set gateway for router l3-router
[root@controller1 ~]# neutron router-list
```

id	name	external_gateway_info	distributed	ha
...86d	l3-router	{"network_id": "f0478328-6d29-4fb7-8d96-5b57f6b6464f",	False	True
		"enable_snat": true, "external_fixed_ips": [{"subnet_id":		
		"8f04fc59-a372-41f7-be90-7e36a84b096e", "ip_address":		
		"192.168.115.200"}]}		

```
//查看路由上新增的端口
[root@controller1 ~]# neutron router-port-list l3-router
```

id	name	mac_address	fixed_ips
...0cc	HA port tenant ...18a793	fa:16:3e:b6:10:ee	{"subnet_id": "dea72fe1-a99c-
			4f85-8901-43b376ca4194",
			"ip_address":
			"169.254.192.2"} }
...28d		fa:16:3e:05:b6:3c	{"subnet_id": "c0921fde-
			8c5b-4780-81a6-5da5e5b007
			d4", "ip_address":
			"192.128.1.1"} }
...be7	HA port tenant ...18a793	fa:16:3e:ac:f6:96	{"subnet_id": "dea72fe1-
			a99c-
			4f85-8901-43b376ca4194",
			"ip_address":
			"169.254.192.1"} }
...509		fa:16:3e:61:53:e8	{"subnet_id":
			"8f04fc59-a372-41f7-be90-
			7e36a84b096e",
			"ip_address": "192.168.
			115.200"} }

当 HA Router 接入 admin-subnet 子网并为其设置 ext-net 外网网关后，其内部新增了两个端口，端口 IP 分别为 admin-subnet 的网关地址和 ext-subnet 的网关地址（ext-subnet 子网可用 IP 地址池的最小地址）。

7) 查看网络节点 qrouter 命名空间接口变化。

//网络节点1的qrouter命名空间 (MasterRouter)

```
[root@network1 ~]# ip netns exec qrouter-fb6fe477-e382-42d8-b303-81ca8621e86d ip
addr show
41: ha-140a8c78-33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue
state UNKNOWN
link/ether fa:16:3e:b6:10:ee brd ff:ff:ff:ff:ff:ff
inet 169.254.192.2/18 brd 169.254.255.255 scope global ha-140a8c78-33
    valid_lft forever preferred_lft forever
inet 169.254.0.1/24 scope global ha-140a8c78-33
    valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:feb6:10ee/64 scope link
    valid_lft forever preferred_lft forever
42: qr-5eeca55-8f: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue state
UNKNOWN
link/ether fa:16:3e:05:b6:3c brd ff:ff:ff:ff:ff:ff
inet 192.128.1.1/24 scope global qr-5eeca55-8f
    valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:fe05:b63c/64 scope link
    valid_lft forever preferred_lft forever
43: qg-c2eaa6cf-e2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UNKNOWN
link/ether fa:16:3e:61:53:e8 brd ff:ff:ff:ff:ff:ff
inet 192.168.115.200/24 scope global qg-c2eaa6cf-e2
    valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:fe61:53e8/64 scope link
    valid_lft forever preferred_lft forever
//网络节点2的qrouter命名空间 (BackupRouter)
[root@network2 ~]# ip netns exec qrouter-fb6fe477-e382-42d8-b303-81ca8621e86d ip
addr show
40: ha-6c44b430-74: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue state
UNKNOWN
link/ether fa:16:3e:ac:f6:96 brd ff:ff:ff:ff:ff:ff
inet 169.254.192.1/18 brd 169.254.255.255 scope global ha-6c44b430-74
    valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:feac:f696/64 scope link
    valid_lft forever preferred_lft forever
41: qr-5eeca55-8f: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue state
UNKNOWN
link/ether fa:16:3e:05:b6:3c brd ff:ff:ff:ff:ff:ff
42: qg-c2eaa6cf-e2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UNKNOWN
link/ether fa:16:3e:61:53:e8 brd ff:ff:ff:ff:ff:ff
```

在步骤2中已经看到 Master Router 位于 network1 节点, Backup Router 位于 network2 节点。因此, network1 网络节点上的 qrouter 命名空间中 qr 和 qg 接口具有 VIP (Project 网络和 External 网络的网关地址), 而 network2 网络节点上的 qrouter 命名空间中 qr 和 qg 接口没有配置任何 IP。

8) 创建测试实例 test-server, 为其绑定 Floating IP。

```
//创建实例
```

```
[root@controller1 ~]# nova boot --image cirros-0.3.4-x86_64 --flavor 1 --nic\
net-id=09d6e2a3-1efe-4561-b6eb-ffe14a4d98ba --key-name admin-key --security-group
default test-server
```

```
//创建Floating IP
```

```
[root@controller1 ~]# neutron floatingip-create ext-net
Created a new floatingip:
```

```
+-----+-----+
| Field | Value |
+-----+-----+
| floating_ip_address | 192.168.115.201 |
+-----+-----+
```

```
//为实例绑定Floating IP
```

```
[root@controller1 ~]# nova floating-ip-associate test-server 192.168.115.201
[root@controller1 ~]# nova list
```

```
+-----+-----+-----+-----+-----+-----+
| ID | Name | Status | Task State | Power State | Networks |
+-----+-----+-----+-----+-----+-----+
| ...9d90 | test-server | ACTIVE | - | Running | admin-net=192.128.1.4,
192.168.115.201 |
+-----+-----+-----+-----+-----+-----+
```

9) 查看 HA Router 端口变化情况以及网络节点 qrouter 命名空间端口变化情况。

```
//控制节点上查看HA Router端口变化
```

```
[root@controller1 ~]# neutron router-port-list l3-router
```

```
+-----+-----+-----+-----+
| id | name | mac_address | fixed_ips |
+-----+-----+-----+-----+
| ...0cc | HA port tenant ...18a793 | fa:16:3e:b6:10:ee | {"subnet_id": "dea72fe1-a99c-4f85-8901-43b376ca4194", "ip_address": "169.254.192.2"} |
| ...28d | | fa:16:3e:05:b6:3c | {"subnet_id": "c0921fde-7d4", "ip_address": "192.128.1.1"} |
| ...be7 | HA port tenant ...18a793 | fa:16:3e:ac:f6:96 | {"subnet_id": "dea72fe1-a99c-4f85-8901-43b376ca4194", "ip_address": "169.254.192.1"} |
| ...509 | | fa:16:3e:61:53:e8 | {"subnet_id": "8f04fc59-a372-41f7-be90-7e36a84b096e", "ip_address": "192.168.115.200"} |
+-----+-----+-----+-----+
```



```

+-----+-----+-----+-----+-----+
//网络节点network1中qrouter端口变化
[root@network1 etc]# ip netns exec qrouter-fb6fe477-e382-42d8-b303-81ca8621e86d ip
addr show
41: ha-140a8c78-33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue state
UNKNOWN
    link/ether fa:16:3e:b6:10:ee brd ff:ff:ff:ff:ff:ff
    inet 169.254.192.2/18 brd 169.254.255.255 scope global ha-140a8c78-33
        valid_lft forever preferred_lft forever
    inet 169.254.0.1/24 scope global ha-140a8c78-33
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:feb6:10ee/64 scope link
        valid_lft forever preferred_lft forever
42: qr-5eeca55-8f: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue state
UNKNOWN
    link/ether fa:16:3e:05:b6:3c brd ff:ff:ff:ff:ff:ff
    inet 192.128.1.1/24 scope global qr-5eeca55-8f
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe05:b63c/64 scope link
        valid_lft forever preferred_lft forever
43: qg-c2eaa6cf-e2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UNKNOWN
    link/ether fa:16:3e:61:53:e8 brd ff:ff:ff:ff:ff:ff
inet 192.168.115.200/24 scope global qg-c2eaa6cf-e2
    valid_lft forever preferred_lft forever
inet 192.168.115.201/32 scope global qg-c2eaa6cf-e2
    valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe61:53e8/64 scope link
        valid_lft forever preferred_lft forever
//网络节点network2中qrouter端口变化
[root@network2 ~]# ip netns exec qrouter-fb6fe477-e382-42d8-b303-81ca8621e86d ip
addr show
40: ha-6c44b430-74: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue state
UNKNOWN
    link/ether fa:16:3e:ac:f6:96 brd ff:ff:ff:ff:ff:ff
    inet 169.254.192.1/18 brd 169.254.255.255 scope global ha-6c44b430-74
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:feac:f696/64 scope link
        valid_lft forever preferred_lft forever
41: qr-5eeca55-8f: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue state
UNKNOWN
    link/ether fa:16:3e:05:b6:3c brd ff:ff:ff:ff:ff:ff
42: qg-c2eaa6cf-e2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UNKNOWN
    link/ether fa:16:3e:61:53:e8 brd ff:ff:ff:ff:ff:ff

```

为实例绑定 Floating IP 后，HA Router 端口并没有任何改变，仍然是两个 HA 接口和两个网关接口。Backup Router 端口也无变化，仍然没有任何 IP，但是 Master Router 的 qg 端口增加了为实例绑定的 Floating IP 地址 192.168.115.201。

10) L3 DNAT 高可用验证。

当外网主机通过实例 Floating IP 访问租户内网实例时, L3 Router 进行 DNAT 地址转换, 因此要测试 HA Router 的 DNAT 高可用性, 只需通过外网主机对实例 Floating IP 一直进行 ping 测试, 然后关闭 Master Router 实例, 并检查 BackupRouter 是否接管故障路由实例和 ping 测试是否中断即可。

```
//再次确认MasterRouter位于network1网络节点, BackupRouter位于network2网络节点
[root@controller1 ~]# neutron l3-agent-list-hosting-router l3-router
```

```
+-----+
| id          | host      | admin_state_up | alive | ha_state |
+-----+
| ...d099f3e5a | network2 | True           | :-)   | standby  |
| ...c15814666 | network1 | True           | :-)   | active   |
+-----+
```

```
//检查对Floating IP的ping测试是否正常
```

```
[root@controller1 ~]# ping 192.168.115.201
```

```
PING 192.168.115.201 (192.168.115.201) 56(84) bytes of data.
```

```
64 bytes from 192.168.115.201: icmp_seq=10 ttl=63 time=81.3 ms
```

```
64 bytes from 192.168.115.201: icmp_seq=11 ttl=63 time=1.98 ms
```

```
64 bytes from 192.168.115.201: icmp_seq=12 ttl=63 time=0.968 ms
```

```
//关闭network1网络节点(模拟MasterRouter故障)
```

```
[root@network1]# systemctl poweroff
```

```
//检查network1节点网络服务是否已经停止
```

```
[root@controller1 ~]# neutron agent-list
```

```
+-----+
| id | agent_type | host | availability_zone | alive | admin_state_up |
+-----+
| ...a526 | Open vSwitch agent | network1 | | xxx | True |
| ...b275 | Open vSwitch agent | network2 | | :-) | True |
| ...84b2 | DHCP agent | network2 | nova | :-) | True |
| ...91aa | DHCP agent | network1 | nova | xxx | True |
| ...3e5a | L3 agent | network2 | nova | :-) | True |
| ...4666 | L3 agent | network1 | nova | xxx | True |
| ...30a9 | Metadata agent | network1 | | xxx | True |
| ...cf49 | Open vSwitch agent | compute2 | | :-) | True |
| ...1b7f | Open vSwitch agent | compute1 | | :-) | True |
| ...c5a7 | Metadata agent | network2 | | :-) | True |
+-----+
```

```
//检查network2是否接管network1的故障路由实例
[root@network2 ~]# ip netns exec qrouter-fb6fe477-e382-42d8-b303-81ca8621e86d ip
  addr show
  .....
41: qr-5eeca55-8f:<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue state
    UNKNOWN
    link/ether fa:16:3e:05:b6:3c brd ff:ff:ff:ff:ff:ff
inet 192.128.1.1/24 scope global qr-5eeca55-8f
    valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:fe05:b63c/64 scope link nodad
    valid_lft forever preferred_lft forever
42: qg-c2eaa6cf-e2:<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
    UNKNOWN
    link/ether fa:16:3e:61:53:e8 brd ff:ff:ff:ff:ff:ff
inet 192.168.115.200/24 scope global qg-c2eaa6cf-e2
    valid_lft forever preferred_lft forever
inet 192.168.115.201/32 scope global qg-c2eaa6cf-e2
    valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:fe61:53e8/64 scope link nodad
    valid_lft forever preferred_lft forever
//再次检查对Floating IP的ping测试是否正常
[root@controller1 ~]# ping 192.168.115.201
PING 192.168.115.201 (192.168.115.201) 56(84) bytes of data.
64 bytes from 192.168.115.201: icmp_seq=1 ttl=63 time=7.02 ms
64 bytes from 192.168.115.201: icmp_seq=2 ttl=63 time=1.40 ms
64 bytes from 192.168.115.201: icmp_seq=3 ttl=63 time=0.941 ms
```

可以看到，当位于 network1 网络节点上的 MasterRouter 出现故障后，network2 网络节点上的 BackupRouter 被提升为新的 MasterRouter，并且全部虚拟 IP（两个网关地址和一个 Floating IP）均迁移至新的 MasterRouter 上，因此外网对实例 Floating IP 的访问不受影响。

11) L3 SNAT 高可用验证。

当实例访问外网（如 Internet）时，L3 Router 对实例数据进行 SNAT 转换，因此要验证 SNAT 功能的高可用性，只需访问实例，并在实例中对 Internet 进行 ping 测试，然后关闭 Master Router 实例，并检查 BackupRouter 是否接管故障路由实例和 ping 测试是否中断即可。

```
//步骤10中对network1关闭并重启之后，MasterRouter已切换到network2节点，不再是network1节点
[root@controller1 ~]# neutron l3-agent-list-hosting-router l3-router
+-----+-----+-----+-----+-----+
| id      | host      | admin_state_up | alive | ha_state |
+-----+-----+-----+-----+-----+
| ...c15814666 | network1 | True           | :- ) | standby  |
| ...d099f3e5a | network2 | True           | :- ) | active   |
+-----+-----+-----+-----+-----+
//从实例test-server访问Internet
[root@test-server ~]# ping www.baidu.com
```

```
PING www.a.shifen.com (61.135.169.121) 56(84) bytes of data.
```

```
64 bytes from 61.135.169.121: icmp_seq=2 ttl=128 time=56.0 ms
```

```
64 bytes from 61.135.169.121: icmp_seq=3 ttl=128 time=38.4 ms
```

```
64 bytes from 61.135.169.121: icmp_seq=4 ttl=128 time=38.1 ms
```

```
//关闭network2网络节点 (模拟MasterRouter故障)
```

```
[root@network2]# systemctl poweroff
```

```
//检查network2节点网络服务是否已经停止
```

```
[root@controller1 ~]# neutron agent-list
```

```
+-----+-----+-----+-----+-----+-----+
|      id | agent_type           | host      | availability_zone | alive | admin_state_up |
+-----+-----+-----+-----+-----+-----+
| ...526 | Open vSwitch agent | network1 |                   | :- ) | True            |
neutron-openvswitch-agent |
| ...275 | Open vSwitch agent | network2 |                   | xxx  | True            |
neutron-openvswitch-agent |
| ...4b2 | DHCP agent         | network2 | nova              | xxx  | True            |
neutron-dhcp-agent        |
| ...1aa | DHCP agent         | network1 | nova              | :- ) | True            |
neutron-dhcp-agent        |
| ...e5a | L3 agent           | network2 | nova              | xxx  | True            |
neutron-l3-agent          |
| ...666 | L3 agent           | network1 | nova              | :- ) | True            |
neutron-l3-agent          |
| ...0a9 | Metadata agent     | network1 |                   | :- ) | True            |
neutron-metadata-agent    |
| ...f49 | Open vSwitch agent | compute2 |                   | :- ) | True            |
neutron-openvswitch-agent |
| ...b7f | Open vSwitch agent | compute1 |                   | :- ) | True            |
neutron-openvswitch-agent |
| ...5a7 | Metadata agent     | network2 |                   | xxx  | True            |
neutron-metadata-agent    |
+-----+-----+-----+-----+-----+-----+
```

```
//检查network1是否接管network2的故障路由实例
```

```
[root@network1 ~]# ip netns exec qrouter-fb6fe477-e382-42d8-b303-81ca8621e86d ip
addr show
```

```
.....
11: 5eeca55-8f:<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue state
UNKNOWN
    link/ether fa:16:3e:05:b6:3c brd ff:ff:ff:ff:ff:ff
inet 192.128.1.1/24 scope global 5eeca55-8f
    valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe05:b63c/64 scope link nodad
    valid_lft forever preferred_lft forever
12: qg-c2eaa6cf-e2:<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UNKNOWN
    link/ether fa:16:3e:61:53:e8 brd ff:ff:ff:ff:ff:ff
inet 192.168.115.200/24 scope global qg-c2eaa6cf-e2
```

```

    valid_lft forever preferred_lft forever
inet 192.168.115.201/32 scope global qq-c2eaa6cf-e2
    valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:fe61:53e8/64 scope link nodad
    valid_lft forever preferred_lft forever
//从实例test-server访问Internet
[root@test-server ~]# ping www.baidu.com
PING www.a.shifen.com (61.135.169.125) 56(84) bytes of data.
64 bytes from 61.135.169.125: icmp_seq=1 ttl=128 time=42.6 ms
64 bytes from 61.135.169.125: icmp_seq=2 ttl=128 time=39.6 ms
64 bytes from 61.135.169.125: icmp_seq=3 ttl=128 time=40.0 ms

```

可以看到，位于 network2 上的 MasterRouter 出现故障后，network1 节点上的 BackupRouter 将自己提升为新的 MasterRouter，并且成功接管故障路由实例上的 VIP，因此 L3 Router 的 SNAT 功能并没有受到任何影响。

9.7 DVR 高可用方案

9.7.1 DVR 高可用部署实现

分布式虚拟路由（Distributed Virtual Router，DVR）通过将计算节点直连到外部网络从而增强实现了传统 Self-Service 网络的高可用性，并从根本上解决了传统 Self-Service 网络流量汇聚造成的带宽瓶颈问题。对于绑定了 Floating IP 的实例，Project 网络与 External 网络之间的路由转发完全由计算节点完成，因而排除了传统 Self-Service 网络单网络节点造成的单点故障和网络性能问题，此外，对于具有 Fixed IP 或 Floating IP 的实例，如果进行 Project 网络之间的通信，则不同 Project 网络之间的路由转发也完全由计算节点完成。但是，对于仅有 Fixed IP 的实例，Project 网络与 External 网络之间的通信仍然依赖于网络节点的路由功能和 SNAT 服务，即 DVR 网络架构中仍然需要网络节点提供三层 SNAT 服务。

与传统 Self-Service 网络架构不同，DVR 网络架构将 L3 Agent 分布到各个计算节点，但是网络节点仍然保留，因此 DVR 网络拓扑与传统 Self-Service 网络拓扑会有较大的不同。为了便于分析，本节以四节点 OpenStack 集群为例，来介绍 DVR 网络架构的部署实现，其中计算节点两个，网络节点和控制节点各一个。在网络接口配置上，控制节点需要一个管理网络接口，此外，为了同时提供 VLAN 类型和 GRE/VxLAN 类型的租户网络，每个计算节点和网络节点需要提供四个网络接口（这不是硬性要求，用户也可以在租户隧道接口和租户 VLAN 接口之间二选一），四个网络接口分别用于管理网络、租户隧道网络、租户 VLAN 网络和外部网络（如 Internet）。其中，租户 VLAN 网络接口将桥接到 OpenvSwitch 网桥 br-vlan 中，外部网络接口将桥接到 OpenvSwitch 网桥 br-ex 中。基于 DVR 网络架构的 OpenStack 集群网络拓扑如图 9-54 所示，各个节点的服务拓扑如图 9-55 所示。

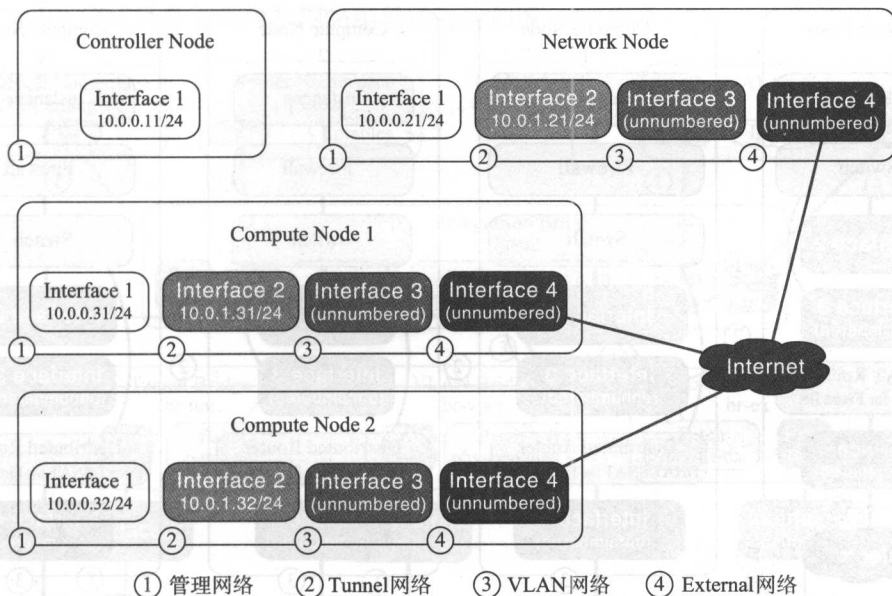


图 9-54 DVR 架构网络拓扑

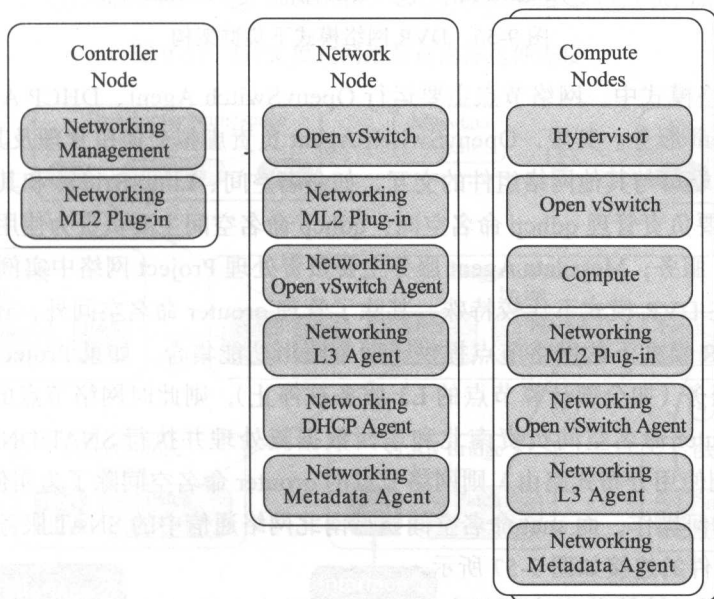


图 9-55 DVR 网络架构节点服务拓扑

基于 DVR 网络模式的 OpenStack 集群架构与传统 Self-Service 网络架构的不同之处主要在于，全部计算节点都直接与外网连接，同时计算节点与网络节点的连接仍然保持不变，DVR 网络模式集群架构如图 9-56 所示。

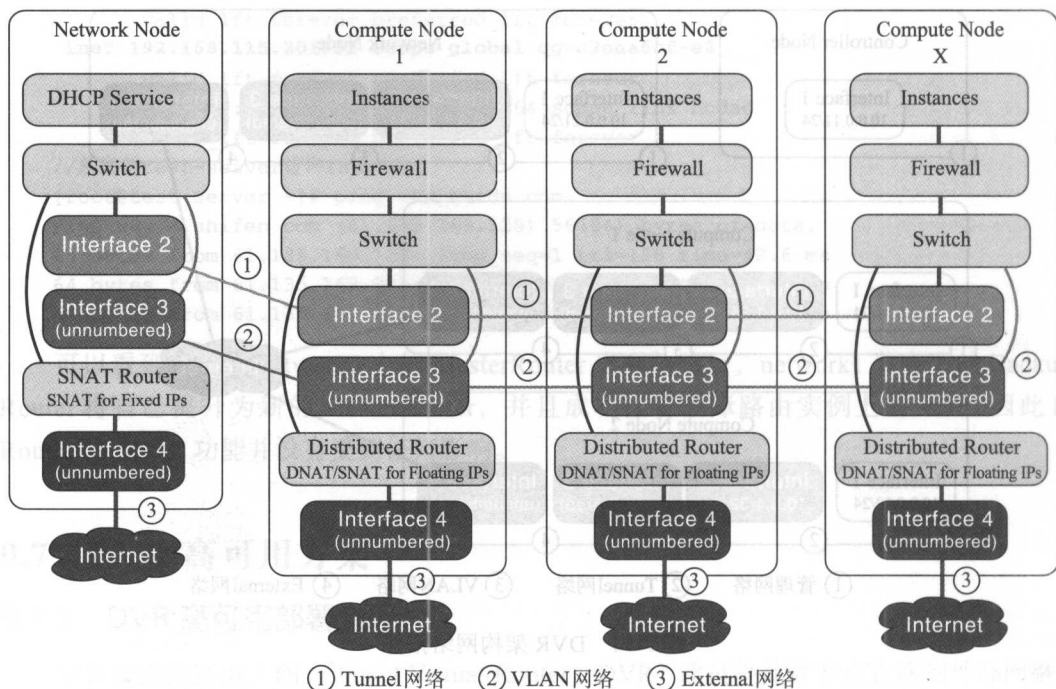


图 9-56 DVR 网络模式下集群架构

在 DVR 网络模式中，网络节点主要运行 OpenvSwitch Agent、DHCP Agent、L3 Agent 和 Metadata Agent 服务。其中，OpenvSwitch Agent 负责虚拟交换机管理及其通信，同时还负责通过其虚拟端口与其他网络组件的交互，如命名空间、LinuxBridge 和其他底层接口等；DHCP Agent 主要负责管理 qdhcp 命名空间，qdhcp 命名空间主要负责为使用 Project 网络的实例提供 DHCP 服务；Metadata Agent 服务主要负责处理 Project 网络中实例的元数据操作；L3 Agent 服务在 DVR 模式下比较特殊，其除了管理 qrouter 命名空间外，还要管理 snat 命名空间，即 DVR 模式下的网络节点提供了两个路由功能集合。如果 Project 网络中的实例使用传统路由服务（如全部计算节点的 L3 服务被停止），则此时网络节点的 snat 命名空间不工作，而 qrouter 命名空间负责南北和东西数据流处理并执行 SNAT/DNAT 转换。如果 Project 网络实例使用分布式路由，则网络节点的 qrouter 命名空间除了为实例元数据操作提供服务外不做任何操作，而 snat 命名空间执行南北网络通信中的 SNAT 服务。DVR 模式下网络节点内部组件的连接如图 9-57 所示。

DVR 模式下，计算节点主要运行 OpenvSwitch Agent、L3 Agent、Metadata Agent 和 Linuxbridge 服务。这里比较关键的就是 L3 Agent 服务，计算节点的 L3 Agent 负责管理 qrouter 和 fip 命名空间，如果 Project 网络中具有 Floating IP 的实例使用分布式路由进行通信，则计算节点 fip 命名空间负责路由南北网络数据流并执行 SNAT 和 DNAT 操作。如果不同 Project 网络中的实例进行彼此通信，则计算节点 qrouter 命名空间负责东西网络数据

流处理。DVR 模式计算节点内部网络组件的连接如图 9-58 所示。

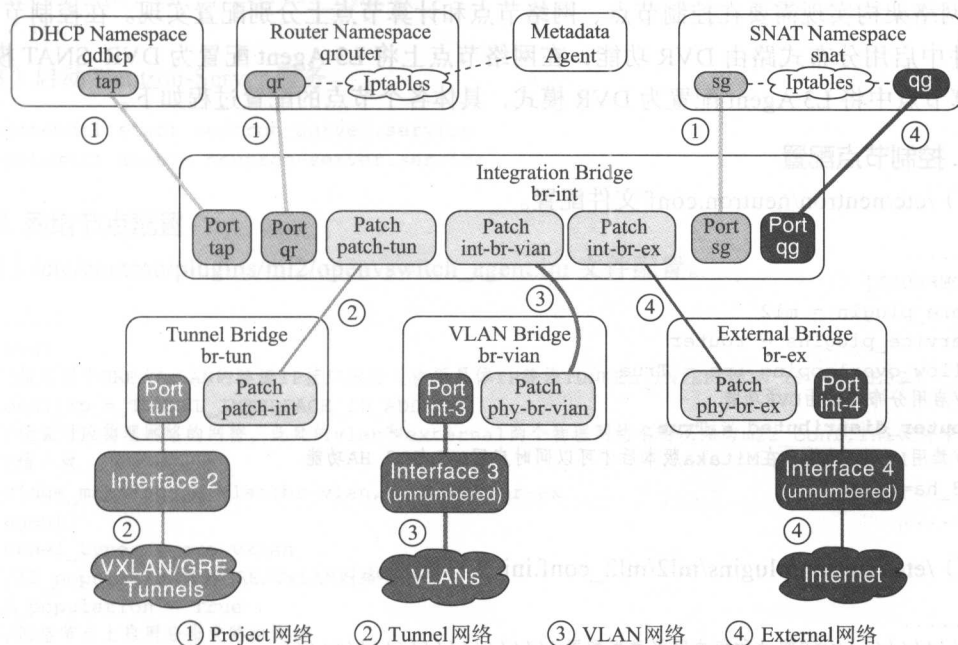


图 9-57 DVR 模式网络节点内部组件连接

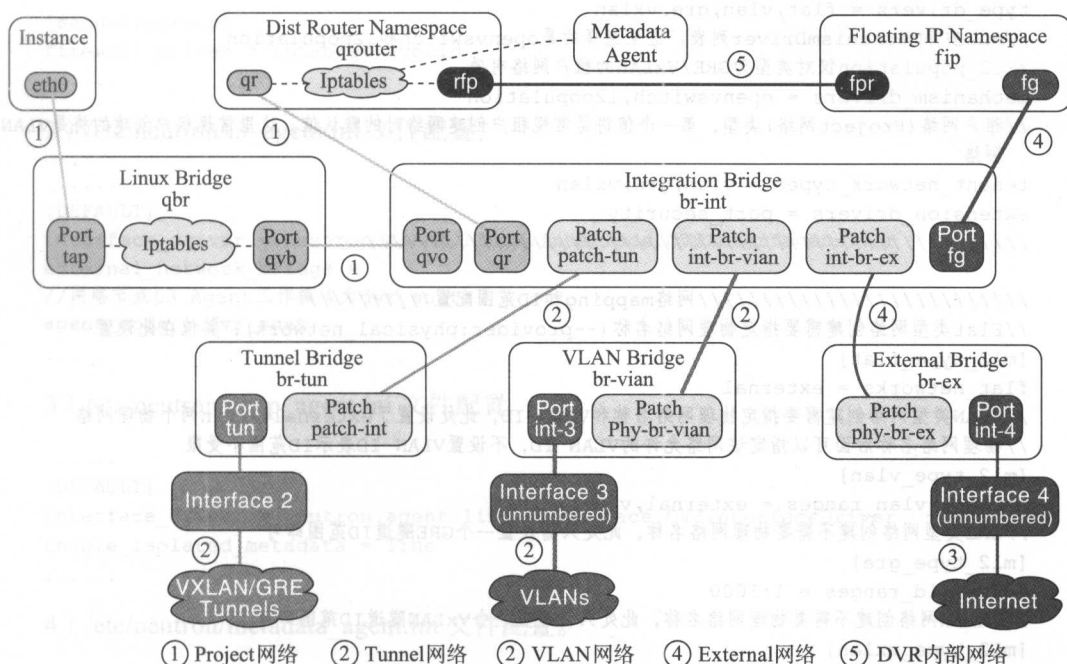


图 9-58 DVR 模式计算节点内部组件连接

DVR 网络模式在配置实现上与传统 Self-Service 或 L3 HA 网络架构有较大的区别, DVR 网络架构实现需要在控制节点、网络节点和计算节点上分别配置实现。在控制节点配置文件中启用分布式路由 DVR 功能, 在网络节点上将 L3 Agent 配置为 DVR_SNAT 模式, 在计算节点中将 L3 Agent 配置为 DVR 模式, 具体各个节点的配置过程如下。

1. 控制节点配置

1) /etc/neutron/neutron.conf 文件配置。

```
.....
[DEFAULT]
core_plugin = ml2
service_plugins = router
allow_overlapping_ips = True
//启用分布式路由DVR功能
router_distributed = True
//禁用L3 HA功能, 在Mitaka版本后才可以同时启用DVR与L3 HA功能
l3_ha=False
.....
```

2) /etc/neutron/plugins/ml2/ml2_conf.ini 文件配置。

```
.....
//////////ML2驱动和租户网络类型配置//////////
[ml2]
//ML2插件TypeDriver列表
type_drivers = flat,vlan,gre,vxlan
//ML2插件MechanismDriver列表, 这里选择的是openvswitch和l2population
//l2_population仅对类型为GRE/VxLAN的租户网络有效
mechanism_drivers = openvswitch,l2population
//租户网络(Project网络)类型, 第一个值将是常规租户创建网络时的默认值, 这里常规租户创建的将是VLAN
网络
tenant_network_types = vlan,gre,vxlan
extension_drivers = port_security
//////////

//////////网络mapping和ID范围配置//////////
//Flat类型网络创建需要指定物理网络名称(--provider:physical_network), 其值在此设置
[ml2_type_flat]
flat_networks = external
//VLAN类型网络创建需要指定物理网络名称和VLAN ID, 此处设置了external和vlan两个物理网络
//物理网络名称后面可以指定该网络允许的VLAN ID, 不设置VLAN ID表示ID范围不受限
[ml2_type_vlan]
network_vlan_ranges = external,vlan:1:1024
//GRE类型网络创建不需要物理网络名称, 此处只需设置一个GRE隧道ID范围即可
[ml2_type_gre]
tunnel_id_ranges = 1:1000
//VxLAN网络创建不需要物理网络名称, 此处只需设置一个VxLAN隧道ID范围即可
[ml2_type_vxlan]
vni_ranges = 1:1000
//////////
```

```
[securitygroup]
firewall_driver = iptables_hybrid
.....
```

3) 启动 Neutron-Server 服务。

```
systemctl start neutron-server.service
systemctl enable neutron-server.service
```

2. 网络节点配置

1) /etc/neutron/plugins/ml2/openvswitch_agent.ini 文件配置。

```
.....
[ovs]
//指定用于GRE/VxLAN的隧道IP接口地址(使用具体IP替换TUNNEL_INTERFACE_IP_ADDRESS)
local_ip = TUNNEL_INTERFACE_IP_ADDRESS
//设置对应物理网络的网桥,此处的vlan和external两个物理网络名称必须与ml2_conf.ini文件中设定
//值一致
bridge_mappings = vlan:br-vlan,external:br-ex
[agent]
tunnel_types = gre,vxlan
//l2_population仅对GRE/VxLAN网络有效
l2_population = True
//网络节点上启用分布式路由支持
enable_distributed_routing = True
arp_responder = True
```

```
[securitygroup]
firewall_driver = iptables_hybrid
.....
```

2) /etc/neutron/l3_agent.ini 文件配置。

```
.....
[DEFAULT]
interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver
external_network_bridge =
//网络节点L3 Agent工作模式为dvr_snat
agent_mode = dvr_snat
.....
```

3) /etc/neutron/dhcp_agent.ini 文件配置。

```
.....
[DEFAULT]
interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver
enable_isolated_metadata = True
.....
```

4) /etc/neutron/metadata_agent.ini 文件配置。

```
.....
[DEFAULT]
```



```
nova_metadata_ip = controller
metadata_proxy_shared_secret = METADATA_SECRET
.....
```

5) 启动网络节点服务。

```
systemctl start openvswitch.service
systemctl enable openvswitch.service
systemctl start neutron-l3-agent.service
systemctl enable neutron-l3-agent.service
systemctl start dhcp-l3-agent.service
systemctl enable dhcp-l3-agent.service
systemctl start openvswitch-l3-agent.service
systemctl enable openvswitch-l3-agent.service
systemctl start metadata-l3-agent.service
systemctl enable metadata-l3-agent.service
```

3. 计算节点配置

1) /etc/neutron/plugins/ml2/openvswitch_agent.ini 文件配置。

```
.....
[ovs]
//使用具体的隧道IP替换TUNNEL_INTERFACE_IP_ADDRESS
local_ip = TUNNEL_INTERFACE_IP_ADDRESS
bridge_mappings = vlan:br-vlan,external:br-ex
[agent]
tunnel_types = gre,vxlan
enable_distributed_routing = True
l2_population = True
arp_responder = True
[securitygroup]
firewall_driver = iptables_hybrid
.....
```

2) /etc/neutron/l3_agent.ini 文件配置。

```
.....
[DEFAULT]
interface_driver = Neutron.agent.linux.interface.OVSInterfaceDriver
external_network_bridge =
//配置计算节点L3 Agent为DVR模式
agent_mode = dvr
.....
```

3) /etc/neutron/plugins/ml2/metadata.ini 文件配置。

```
.....
[DEFAULT]
nova_metadata_ip = controller
metadata_proxy_shared_secret = METADATA_SECRET
.....
```

4) 启动计算节点网络服务。

```
systemctl start openvswitch.service
systemctl enable openvswitch.service
systemctl start neutron-openvswitch-agent.service
systemctl enable neutron-openvswitch-agent.service
systemctl start neutron-l3-agent.service
systemctl enable neutron-l3-agent.service
systemctl start metadata-l3-agent.service
systemctl enable metadata-l3-agent.service
```

各个节点网络服务配置并启动完成之后,在控制节点加载管理员权限,验证 DVR 模式下各个节点 Neutron 代理服务启动情况。正常情况下,各个节点网络代理服务运行情况应该如下:

```
[root@controller1 ~]# neutron agent-list
```

id	agent_type	host	alive	admin_state_up	binary
...6bea	Metadata agent	computel	:-)	True	neutron-metadata-agent
...7529	Metadata agent	compute2	:-)	True	neutron-metadata-agent
...a526	Open vSwitch agent	network1	:-)	True	neutron-openvswitch-agent
...91aa	DHCP agent	network1	:-)	True	neutron-dhcp-agent
...4666	L3 agent	network1	:-)	True	neutron-l3-agent
...30a9	Metadata agent	network1	:-)	True	neutron-metadata-agent
...cf49	Open vSwitch agent	compute2	:-)	True	neutron-openvswitch-agent
...2cf1	L3 agent	computel	:-)	True	neutron-l3-agent
...247e	L3 agent	compute2	:-)	True	neutron-l3-agent
...1b7f	Open vSwitch agent	computel	:-)	True	neutron-openvswitch-agent

9.7.2 DVR 高可用验证与分析

1. 网络与分布式路由创建

1) 首先创建 External 网络 ext-net 及其子网 ext-subnet。

本节将创建 Flat 类型的 External 网络和 GRE 类型的 Project 网络,用户也可以选择创建 VLAN 类型的 External 网络和 VxLAN 或 VLAN 类型的 Project 网络。ext-subnet 子网可

用 IP 地址段为 192.168.115.200~192.168.115.250, 外部网络禁止使用 DHCP 功能。

```
//创建外网ext-net
[root@controller1 ~]# neutron net-create ext-net --router:external True \
    --provider:physical_network external--provider:network_type flat
//创建外网子网ext-subnet
[root@controller1 ~]# neutron subnet-create ext-net 192.168.115.0/24 --name ext-subnet
    --allocation-pool start=192.168.115.200,end=192.168.115.250 --disable-dhcp
    --gateway 192.168.115.254
[root@controller1 ~]# neutron net-list
```

id	name	subnets
...48af69b5	ext-net	1be010f3-c5f2-42a4-87db-ccd64c56b70e 192.168.115.0/24

2) 创建 Project 网络 admin-net 及其子网 admin-subnet。

admin-subnet 为租户子网, 租户可以自定义子网名称及其 IP 地址范围, 每个租户可以创建多个租户子网, 为了便于后续网络分析, 此处创建两个不同的租户网络及其子网: admin-net1 及其子网 admin-subnet1, admin-net2 及其子网 admin-subnet2。

```
[root@controller1 ~]# neutron net-create admin-net1 --provider:network_type gre
[root@controller1 ~]# neutron net-create admin-net2 --provider:network_type gre
[root@controller1 ~]# neutron subnet-create admin-net1 --name admin-subnet1
    --gateway 192.128.1.1 192.128.1.0/24
[root@controller1 ~]# neutron subnet-create admin-net2 --name admin-subnet2
    --gateway 192.128.2.1 192.128.2.0/24
[root@controller1 ~]# neutron net-list
```

id	name	subnets
...69b5	ext-net	1be010f3-c5f2-42a4-87db-ccd64c56b70e 192.168.115.0/24
...5416	admin-net1	07cf2823-e41b-48de-a0cc-2bac0faba4e4 192.128.1.0/24
...a77a	admin-net2	2c7e06bd-f7e9-4070-8c2b-2d7a84006519 192.128.2.0/24

```
[root@controller1 ~]# neutron subnet-list
```

id	name	cidr	allocation_pools
...b70e	ext-subnet	192.168.115.0/24	{"start": "192.168.115.200", "end": "192.168.115.250"}
...a4e4	admin-subnet1	192.128.1.0/24	{"start": "192.128.1.2", "end": "192.128.1.254"}
...6519	admin-subnet2	192.128.2.0/24	{"start": "192.128.2.2", "end": "192.128.2.254"}

3) 分布式路由创建前检查计算节点与网络节点是否有 qrouter 命名空间。

```
[root@computel ~]# ip netns
```

```
[root@compute2 ~]# ip netns
[root@network1 ~]# ip netns
qdhcp-6c1dec69-cdc8-4277-91a2-63511e4581a8
```

4) 创建分布式虚拟路由 dvr-router。

```
[root@controller1 ~]# neutron router-create dvr-router
Created a new router:
```

Field	Value
distributed	True

```
[root@controller1 ~]# neutron router-list
```

id	name	external_gateway_info
distributed ha		
20cec0bd-117e-45d2-8565-aac185a51fe0	dvr-router	null
True	False	

5) 为分布式路由 dvr-router 设置两个 Project 网络接口和 External 网络网关, 并检查计算节点是否生成 qrouter 和 fip 命名空间, 网络节点是否生成 snat 和 qrouter 命名空间。

```
[root@controller1 ~]# neutron router-interface-add dvr-router admin-subnet1
Added interface 2250c6d0-cb40-402a-9dbc-2f7140bca43b to router dvr-router.
[root@controller1 ~]# neutron router-interface-add dvr-router admin-subnet2
Added interface 8871516b-b100-4154-b95b-b1a9146118c1 to router dvr-router.
[root@controller1 ~]# neutron router-gateway-set dvr-router ext-net
Set gateway for router dvr-router
[root@controller1 ~]# neutron router-list
```

id	name	external_gateway_info
distributed ha		
...51fe0	dvr-router	{"network_id": "b59c846a-6959-48df-ba81-5f0e48af69b5",
True	False	"enable_snat": true, "external_fixed_ips": [{"subnet_id":
		"1be010f3-c5f2-42a4-87db-ccd64c56b70e", "ip_address":
		"192.168.115.200"]}]}

```
//检查网络节点是否有snat和qrouter命名空间
[root@network1 ~]# ip netns
snat-20cec0bd-117e-45d2-8565-aac185a51fe0
qrouter-20cec0bd-117e-45d2-8565-aac185a51fe0
qdhcp-6c1dec69-cdc8-4277-91a2-63511e4581a8
//检查两个计算节点是否有fip和qrouter命名空间
[root@compute1 ~]# ip netns
[root@compute2 ~]# ip netns
```

可以看到,分布式路由创建之后,网络节点对应生成了 snat 和 qrouter 命名空间,而两个计算节点仍然没有生成 qrouter 和 fip 命名空间。

6) 在计算节点 compute1 和 compute2 上分别创建两个实例 dvr-server1 和 dvr-server2。

```
[root@controller1 ~]# nova boot --flavor 1 --image cirros-0.3.4-x86_64 --key-name
admin-key --security-group default --nic net-id= 748d632a-3235-4313-87f8-
f6745e0c5416 dvr-server1
[root@controller1 ~]# nova boot --flavor 1 --image cirros-0.3.4-x86_64 --key-name
admin-key --security-group default --nic net-id= 262e09f3-ae8d-48c7-bba3-
ff62f15da77a dvr-server2
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State	Networks
...894f3b7d8	dvr-server2	ACTIVE	-	Running	admin-net2=192.128.2.5
...075e403f2	dvr-server1	ACTIVE	-	Running	admin-net1=192.128.1.4

//dvr-server1位于compute1节点

```
[root@controller1 ~]# nova hypervisor-servers compute1
```

ID	Name	Hypervisor ID	Hypervisor Hostname
...075e403f2	instance-00000024	1	compute1

//dvr-server2位于compute2节点

```
[root@controller1 ~]# nova hypervisor-servers compute2
```

ID	Name	Hypervisor ID	Hypervisor Hostname
...894f3b7d8	instance-00000026	2	compute2

7) 实例创建后,检查计算节点是否生成 qrouter 和 fip 命名空间。

//计算节点compute1命名空间

```
[root@compute1 ~]# ip netns
qrouter-20cec0bd-117e-45d2-8565-aac185a51fe0
```

//计算节点compute2命名空间

```
[root@compute2 neutron]# ip netns
qrouter-20cec0bd-117e-45d2-8565-aac185a51fe0
```


//网络节点命名空间

```
[root@network1 ~]# ip netns
snat-20cec0bd-117e-45d2-8565-aac185a51fe0
qrouter-20cec0bd-117e-45d2-8565-aac185a51fe0
qdhcp-98d63ac7-6696-48fb-8863-e68b2dbba87f
```

8) 创建两个 Floating IP 并分别关联到 dvr-server1 和 dvr-server2。

```
[root@controller1 ~]# neutron floatingip-create ext-net
Created a new floatingip:
```

Field	Value
floating_ip_address	192.168.115.202

```
[root@controller1 ~]# neutron floatingip-create ext-net
Created a new floatingip:
```

Field	Value
floating_ip_address	192.168.115.203

```
[root@controller1 ~]# nova floating-ip-associate dvr-server1 192.168.115.202
```

```
[root@controller1 ~]# nova floating-ip-associate dvr-server2 192.168.115.203
```

```
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State	Networks
...075e403f2	dvr-server1	ACTIVE	-	Running	admin-net=192.128.1.4, 192.168.115.202
...894f3b7d8	dvr-server2	ACTIVE	-	Running	admin-net=192.128.2.5, 192.168.115.203

9) 实例绑定 Floating IP 后，检查计算节点和网络节点命名空间变化。

//compute1命名空间

```
[root@compute1 ~]# ip netns
fip-50e216e0-b08b-4209-8642-d444d83de059
qrouter-20cec0bd-117e-45d2-8565-aac185a51fe0
```

//compute2命名空间

```
[root@compute2 ~]# ip netns
fip-50e216e0-b08b-4209-8642-d444d83de059
qrouter-20cec0bd-117e-45d2-8565-aac185a51fe0
```

//网络节点network1命名空间

```
[root@network1 ~]# ip netns
snat-20cec0bd-117e-45d2-8565-aac185a51fe0
qrouter-20cec0bd-117e-45d2-8565-aac185a51fe0
qdhcp-98d63ac7-6696-48fb-8863-e68b2dbba87f
```

可以看到,在创建分布式路由后和实例创建之前,计算节点不会生成 `qrouter` 和 `fip` 命名空间,而网络节点会生成 `qrouter` 和 `snat` 命名空间。当实例创建后,对应实例宿主机的计算节点上会生成 `qrouter` 命名空间,而只有在为实例绑定 Floating IP 之后,对应的计算节点上才会出现 `fip` 命名空间,上述过程创建的 DVR 网络拓扑如图 9-59 所示。图 9-59 所示中, `dvr-server1` 接入 `admin-net1` 租户网络, `dvr-server2` 接入 `admin-net2` 租户网络,而两个租户网络同时接入分布式路由 `dvr-router`。

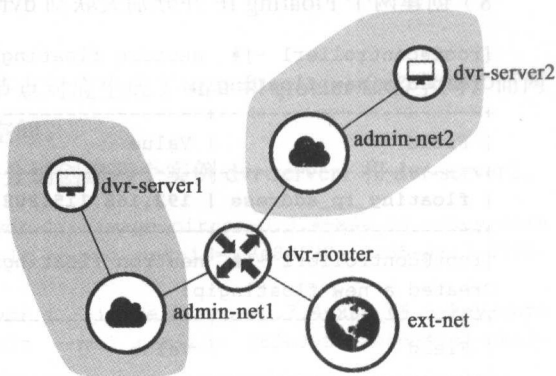


图 9-59 DVR 模式下不同 Project 网络实例拓扑

在 Juno 版本之前, OpenStack 集群网络负载几乎完全由网络节点来承担,即使位于相同宿主机上的两个实例彼此之间需要通信,实例之间的数据流也要经过网络节点 L3 Service 的处理,这极易造成网络节点的工作负载,同时网络节点形成了 OpenStack 集群的单点故障。Juno 版本发行后, Neutron 支持的 DVR 网络架构解决了这一问题,在 DVR 网络架构中, L3 Service 提供三种类型的服务:

- ❑ East-West 通信服务。即相同租户不同子网中实例之间的网络通信。
- ❑ DNAT 服务。外网通过 Floating IP 对数据中心租户实例进行直接访问,这是 DNAT 功能实现的南北网络通信。
- ❑ SNAT 服务。数据中心未绑定 Floating IP 实例通过共享外网网关地址对外部网络进行访问,这是 SNAT 功能实现的南北网络通信。

DVR 网络架构下,计算节点中的 L3 Agent 实现了前两类通信服务,从而将网络节点的负载分流并降低了网络节点单点故障的影响。但是 L3 Service 的 SNAT 通信服务仍然依赖传统集中式部署的网络节点来实现, OpenStack 集群中采用 SNAT 方式进行通信的实例数据流仍然汇聚到网络节点,并使得网络节点对 L3 SNAT 功能构成单点故障。在 DVR 模式下,要消除网络节点 SNAT 功能可能造成的性能瓶颈和单点故障问题,用户可以为全部需要进行南北网络通信的实例绑定 Floating IP,使得全部实例对外网的访问都由本地计算节点的 L3 Agent 实现,从而将网络节点功能闲置。DVR 模式下 L3 Service 的服务分布如图 9-60 所示。

图 9-60 所示中, L3 Service 插件部署在控制节点上,计算节点的 L3 Agent 运行在 DVR 模式,网络节点的 L3 Agent 运行在 DVR_SNAT 模式。计算节点运行在 DVR 模式的 L3 Agent 通过 Linux 网络命名空间实现虚拟路由设备,其实现过程本质上是对传统中心化网络节点 L3 Agent 所实现的虚拟路由的克隆。因此,每个计算节点上的 `qrouter` 命名空间完全一致,命名空间接口 MAC 和 IP 地址也完全相同。以本节创建的分布式路由 `dvr-router`,租户

网络 admin-net1 和 admin-net2 为例, 计算节点 compute1 和 compute2 中 qrouter 命名空间接口如下:

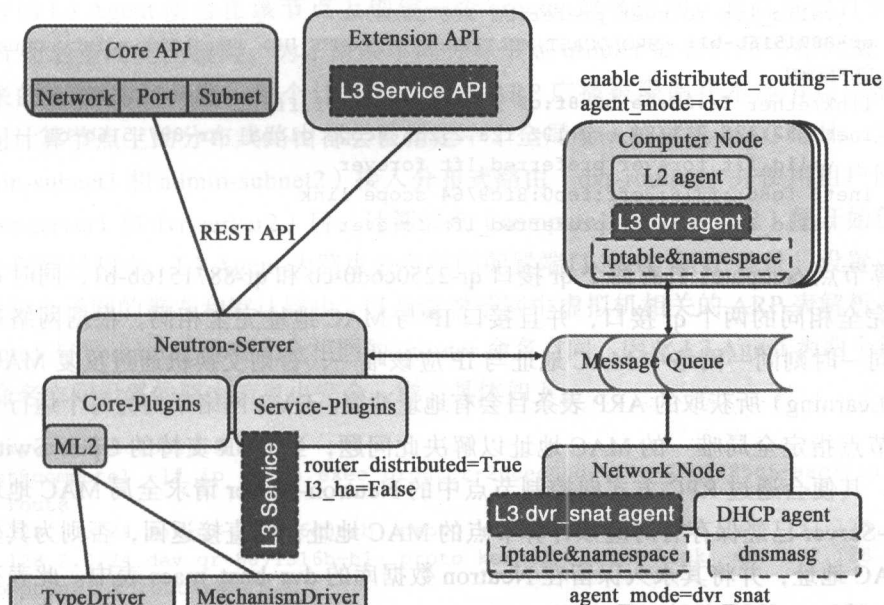


图 9-60 DVR 模式下 L3 Service 分布情况

//compute1路由命名空间接口

```
[root@compute1 ~]# ip netns exec qrouter-20cec0bd-117e-45d2-8565-aac185a51fe0 ip
addr show
```

```
47: qr-2250c6d0-cb: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue state
UNKNOWN
```

```
link/ether fa:16:3e:4d:15:43 brd ff:ff:ff:ff:ff:ff
```

```
inet 192.128.1.1/24 brd 192.128.1.255 scope global qr-2250c6d0-cb
```

```
valid_lft forever preferred_lft forever
```

```
inet6 fe80::f816:3eff:fe4d:1543/64 scope link
```

```
valid_lft forever preferred_lft forever
```

```
48: qr-8871516b-b1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue state
UNKNOWN
```

```
link/ether fa:16:3e:b0:8f:c9 brd ff:ff:ff:ff:ff:ff
```

```
inet 192.128.2.1/24 brd 192.128.2.255 scope global qr-8871516b-b1
```

```
valid_lft forever preferred_lft forever
```

```
inet6 fe80::f816:3eff:feb0:8fc9/64 scope link
```

```
valid_lft forever preferred_lft forever
```

//compute1路由命名空间接口

```
[root@compute1 ~]# ip netns exec qrouter-20cec0bd-117e-45d2-8565-aac185a51fe0 ip
addr show
```

```
40: qr-2250c6d0-cb: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue
state UNKNOWN
```

```
link/ether fa:16:3e:4d:15:43 brd ff:ff:ff:ff:ff:ff
```

```

inet 192.128.1.1/24 brd 192.128.1.255 scope global qr-2250c6d0-cb
    valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:fe4d:1543/64 scope link
    valid_lft forever preferred_lft forever
43: qr-8871516b-b1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue state
    UNKNOWN
    link/ether fa:16:3e:b0:8f:c9 brd ff:ff:ff:ff:ff:ff
inet 192.128.2.1/24 brd 192.128.2.255 scope global qr-8871516b-b1
    valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:feb0:8fc9/64 scope link
    valid_lft forever preferred_lft forever

```

计算节点 `compute1` 具有两个 `qr` 接口 `qr-2250c6d0-cb` 和 `qr-8871516b-b1`，同时 `compute2` 也具有完全相同的两个 `qr` 接口，并且接口 IP 与 MAC 地址完全相同。根据网络基础知识可知，同一时刻同一网络中 MAC 地址与 IP 应该唯一，否则交换机通过反复 MAC 地址学习（Re-Learning）所获取的 ARP 表条目会有地址冲突。DVR 网络架构为每个运行 L3 Agent 的计算节点指定全局唯一的 MAC 地址以解决此问题，当 DVR 支持的 OpenvSwitch Agent 启动时，其便会通过 RPC 方式向控制节点中的 Neutron-Server 请求全局 MAC 地址。如果 Neutron-Server 已经保存有对应该计算节点的 MAC 地址，则直接返回，否则为其生成一个新的 MAC 地址，并将其永久保留在 Neutron 数据库的 `dvr_host_macs` 表中。此表存储的数据格式如下：

```

[root@controller1 ~]# mysql -uroot -proot -e "use neutron;select * from dvr_host_macs;"

```

```

+-----+-----+
| host      | mac_address |
+-----+-----+
| network1  | fa:16:3f:80:c4:ab |
| compute2  | fa:16:3f:d2:27:d4 |
| compute1  | fa:16:3f:d6:20:1c |
+-----+-----+

```

计算节点 OpenvSwitch Agent 申请到主机 MAC 地址后，位于其上的任何虚拟机实例在访问其他计算节点上的实例时，此计算节点上路由命名空间接口源 MAC 地址都会在 OVS 网桥上被申请到的主机 MAC 地址所替换。而当数据包进入计算节点后，如果数据包中网络标识与本地虚拟机所属网络匹配，并且数据包中目的 MAC 地址也与本地虚拟机的 MAC 地址匹配，则该数据包中来自远程计算节点的主机 MAC 地址就会被本地虚拟机网络的网关 MAC 地址替换。DVR 网络架构下，计算节点分布式路由对不同形式的实例通信（East-West/South-North）采取不同的处理方式，下面分别对实例数据的东西和南北网络数据通信进行分析。

2. DVR 模式东西数据流分析

在 DVR 模式中，当租户创建了一个分布式 Router，同时为其设置了租户网络接口和

外网网关，并在计算节点创建虚拟机后（仅创建分布式 Router 而没有创建实例，则只有网络节点生成路由命名空间，计算节点并不会生成路由命名空间），对应计算节点上以 DVR 模式运行的 L3 Agent 便会在该节点上创建一个 qrouter 命名空间，并且各个计算节点上的 qrouter 命名空间完全相同。为了解决不同计算节点 qrouter 命名空间中相同 IP 和 MAC 地址带来的 ARP 冲突问题，每个计算节点上的 ARP 广播被限制在本地节点内部，并且位于不同计算节点上的分布式路由都会被指定一个全局唯一的主机 MAC 地址。当租户子网（admin-subnet1 和 admin-subnet2）接入分布式路由（dvr-router），并使用租户网络创建实例（dvr-server1 和 dvr-server2）后，计算节点（compute1 和 compute2）便开始创建虚拟路由。在创建过程中，L3 Agent 为路由命名空间配置端口 MAC 和 IP 地址、设置子网路由表（包括每个子网的静态和默认路由）以及设置子网中虚拟机相关的 ARP 表解析条目。由于 compute1 与 compute2 具有完全相同的 qrouter 命名空间，因此 L3 Agent 为两个计算节点 qrouter 命名空间设置的路由信息也完全一样，具体如下：

```
//compute1节点qrouter路由信息
[root@compute1 ~]# ip netns exec qrouter-20cec0bd-117e-45d2-8565-aac185a51fe0 ip
route
192.128.1.0/24 dev qr-2250c6d0-cb proto kernel scope link src 192.128.1.1
192.128.2.0/24 dev qr-8871516b-b1 proto kernel scope link src 192.128.2.1
//compute2节点qrouter路由信息
[root@compute2 ~]# ip netns exec qrouter-20cec0bd-117e-45d2-8565-aac185a51fe0 ip
route
192.128.1.0/24 dev qr-2250c6d0-cb proto kernel scope link src 192.128.1.1
192.128.2.0/24 dev qr-8871516b-b1 proto kernel scope link src 192.128.2.1
```

从节点 qrouter 中的路由信息可以看到，目标网络为 192.128.2.0/24 的数据包经过 qr-8871516b-b1 接口转发，对应数据包中源 MAC 地址也会被替换为 qr-8871516b-b1 端口 MAC 地址。而目标网络为 192.128.2.0/24 的数据包经过 qr-2250c6d0-cb 转发，同样数据包中源 MAC 地址也会被替换为 qr-2250c6d0-c 端口 MAC 地址。

现假设位于 admin-subnet1 租户子网中（gateway:192.168.1.1）的 dvr-server1（Fixed IP:192.168.1.4）访问 admin-subnet2 中（gateway:192.168.2.1）的实例 dvr-server（Fixed IP:192.168.2.5）。admin-subnet1 子网和 admin-subnet2 子网均连接到同一个路由 dvr-router 上，因此路由内部对应配置了两个子网网关端口（qr-2250c6d0-cb 和 qr-8871516b-b1，IP 分别为 192.168.1.1 和 192.168.2.1）。dvr-server1 与 dvr-server2 之间的网络东西网络通信过程如图 9-61 所示。

图 9-61 所示中，源自 admin-subnet1 子网中 dvr-server1 的数据包首先到达 compute1 计算节点 qrouter 命名空间中 qr-2250c6d0-cb 端口，然后根据数据包中的目的 IP 和命名空间中的 ARP 表信息，虚拟路由为数据包设置目的主机 MAC 地址，同时将源主机 MAC 地址替换为路由命名空间中 qr-8871516b-b1 端口的 MAC 地址，然后数据包进入隧道网桥 br-tun，br-tun 将数据包中源 MAC 地址再次替换成 Neutron-Server 为 compute1 分配的全局唯一主机 MAC 地址。之后数据包进入 compute2 计算节点的隧道网桥 br-tun，br-tun 将数据包上的

Tunnel ID 替换为本地 VLAN ID, 并将源 MAC 地址与全局唯一主机 MAC 地址匹配的数据包转发到集成网桥 br-int。br-int 将数据包中 compute1 主机 MAC 地址替换为本地路由命名空间中目标子网端口 (qr-8871516b-b1) 地址, 并将数据包直接转发到 dvr-server2。从二层 MAC 地址变换过程分析, 在 dvr-server1 的数据包到达 dvr-server2 的过程中, 数据包中的源 MAC 地址首先被 compute1 中 qrouter 内 admin-subnet2 接口 MAC 地址替换, 之后到达 br-tun 网桥后又被 compute1 主机 MAC 替换, 进入 compute2 并到达 br-int 网桥后, 再被 compute2 中 qrouter 内 admin-subnet2 接口 MAC 地址替换, 之后数据包进入 admin-subnet2 中的 dvr-server2。

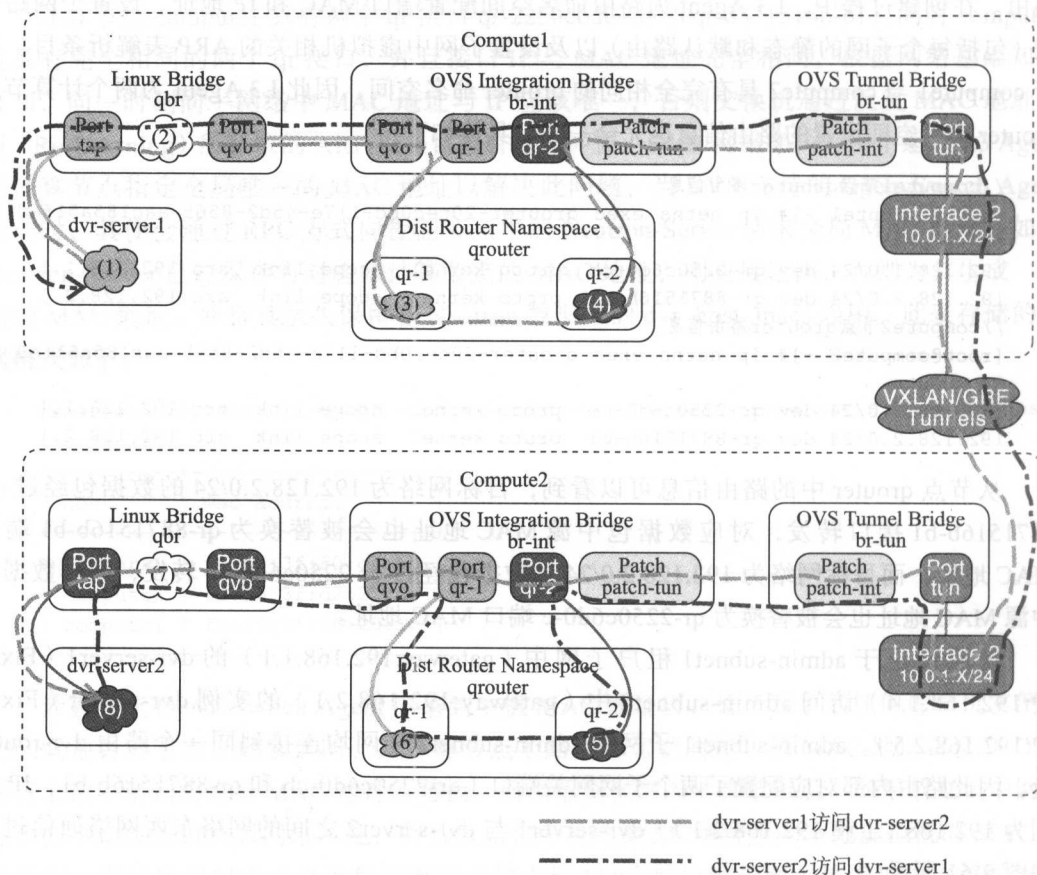


图 9-61 DVR 模式实例东西网络通信

从 DVR 的东西数据流分析看, compute1 中的 dvr-server1 访问 compute2 中的 dvr-server2 时, 仅有 compute1 中的 qrouter 命名空间起到路由转发功能, 而 compute2 中的 qrouter 和网络节点上的 qrouter 都没有工作。相反, 如果 compute2 中的 dvr-server2 访问 compute1 中的 dvr-server1, 则只有 compute2 中的 qrouter 工作, compute1 中的 qrouter 和网络节点中的

qrouter 均不工作。由于 East-West 数据通信中正向访问与反向访问并不经过同一个虚拟路由命名空间，因此无法创建有状态的 East-West Rules，所以基于 East-West 数据流的 FWaaS 高级服务在 DVR 中不被支持。

3. DVR 模式 Floating IP 实例南北数据流分析

实例与外网之间的南北通信可以通过 DNAT 或 SNAT 实现，如果外网 IP 比较丰富，则建议通过 DNAT 方式进行，因为 DNAT 方式下实例数据流直接由计算节点进入外网，而 SNAT 却要汇聚至网络节点才能进入外网。此外，当实例配置有 Floating IP 后，外网便可直接由计算节点访问实例。在 DVR 模式下，当用户为实例绑定 Floating IP 时，计算节点 L3 Agent 将会在 qrouter 命名空间中进行以下操作：

- 1) 在计算节点创建 fip-<netid> 命名空间。
- 2) 在计算节点 qrouter-<routerid> 命名空间中创建 rfp-<portid> 端口。
- 3) 在 rfp 端口上配置租户创建的外网 Floating IP。
- 4) 在 fip 命名空间中创建 fpr 端口，并将其直连到 qrouter-<routerid> 命名空间中的 rfp 接口。fip 命名空间的 fpr 接口与 qrouter 命名空间的 rfp 端口使用默认的内部网络进行二层直连通信。
- 5) 为 fip 命名空间中的网关接口 fg-<portid> 分配一个外网 IP，此端口通常会占用一个额外的外网地址。
- 6) 将 fg-<portid> 端口配置成为 ARP 代理接口，fg-<portid> 端口响应实例 Floating IP 地址的任何 ARP 请求。

具有 Floating IP 的实例进行租户网络与外网之间的南北网络通信时，数据流的处理可以分为两种情况，即外网访问租户实例和租户实例访问外网。现将 compute1 上的 dvr-server1 配置一个外网 Floating IP (192.168.115.203)，则可以观察到 compute1 计算节点上 qrouter 和 fip 命名空间中端口变化情况如下：

```
//qrouter命名空间
[root@compute1 ~]# ip netns exec qrouter-20cec0bd-117e-45d2-8565-aac185a51fe0 ip
addr
2: rfp-20cec0bd-1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP qlen 1000
link/ether a2:a0:95:6d:aa:12 brd ff:ff:ff:ff:ff:ff link-netnsid 0
inet 169.254.31.28/31 scope global rfp-20cec0bd-1
valid_lft forever preferred_lft forever
inet 192.168.115.202/32 brd 192.168.115.202 scope global rfp-20cec0bd-1
valid_lft forever preferred_lft forever
inet6 fe80::a0a0:95ff:fe6d:aal2/64 scope link
valid_lft forever preferred_lft forever
15: qr-2250c6d0-cb: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue
state UNKNOWN
link/ether fa:16:3e:4d:15:43 brd ff:ff:ff:ff:ff:ff
inet 192.128.1.1/24 brd 192.128.1.255 scope global qr-2250c6d0-cb
```

```

    valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe4d:1543/64 scope link
    valid_lft forever preferred_lft forever
16: qr-8871516b-b1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue
    state UNKNOWN
    link/ether fa:16:3e:b0:8f:c9 brd ff:ff:ff:ff:ff:ff
    inet 192.128.2.1/24 brd 192.128.2.255 scope global qr-8871516b-b1
    valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:feb0:8fc9/64 scope link
    valid_lft forever preferred_lft forever
//fip命名空间
[root@compute1 ~]# ip netns exec fip-b59c846a-6959-48df-ba81-5f0e48af69b5 ip addr
2: fpr-20cec0bd-1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
    state UP qlen 1000
    link/ether 36:c3:2c:c4:9c:e9 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 169.254.31.29/31 scope global fpr-20cec0bd-1
    valid_lft forever preferred_lft forever
    inet6 fe80::34c3:2cff:fec4:9ce9/64 scope link
    valid_lft forever preferred_lft forever
17: fg-elb88302-1d: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    state UNKNOWN
    link/ether fa:16:3e:c0:54:50 brd ff:ff:ff:ff:ff:ff
    inet 192.168.115.204/24 brd 192.168.115.255 scope global fg-elb88302-1d
    valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fec0:5450/64 scope link
    valid_lft forever preferred_lft forever

```

为实例 dvr-server1 绑定 Floating IP 192.168.115.202 后, compute1 节点上的 L3 Agent 在 qrouter 命名空间中创建了 rfp-20cec0bd-1 端口, 并配置了默认的内网 IP (169.254.31.28) 和实例 Floating IP (192.168.115.202)。此外, L3 Agent 还新创建了 fip 命名空间, 并在其中配置了 fpr-20cec0bd-1 和 fg-elb88302-1d 端口, 其中 fpr-20cec0bd-1 是与 qrouter 命名空间 rfp-20cec0bd-1 端口进行直连通信的内部接口 (IP 地址为 169.254.31.29), fg-elb88302-1d 是 fip 的外网网关接口, 其上配置了外网 IP (192.168.115.204), fip 命名空间 fg-elb88302-1d 端口会响应全部针对 qrouter 命名空间实例 Floating IP 192.168.115.202 的 ARP 请求, 因此通常认为 fip 的 fg 端口是实例 Floating IP 的 ARP 代理接口。

现假设 compute1 中的 dvr-server1 (FixedIP:192.168.1.4, FloatingIP:192.168.115.202) 访问外网 (192.168.115.0/24) 中的主机, 则实例数据流如图 9-62 所示。dvr-server1 数据进入 qbr 网桥, qbr 对数据进行安全处理后转发到集成网桥 br-int, br-int 再将数据转发到 qrouter 命名空间的 qr 接口 (admin-subnet1 网关接口, IP 为 192.168.1.1), qrouter 利用 rfp 接口上的 FloatingIP (192.168.115.202) 作为源地址对数据包进行 SNAT 转换, qrouter 命名空间通过内部接口 rfp (IP 为 169.254.31.28) 将数据包转发到 fip 命名空间的 fpr 端口 (IP 为 169.254.31.29), fip 命名空间外网网关接口 fg 将数据包转发到外网网桥 br-ex, br-ex 将数据包转发到外部网络。

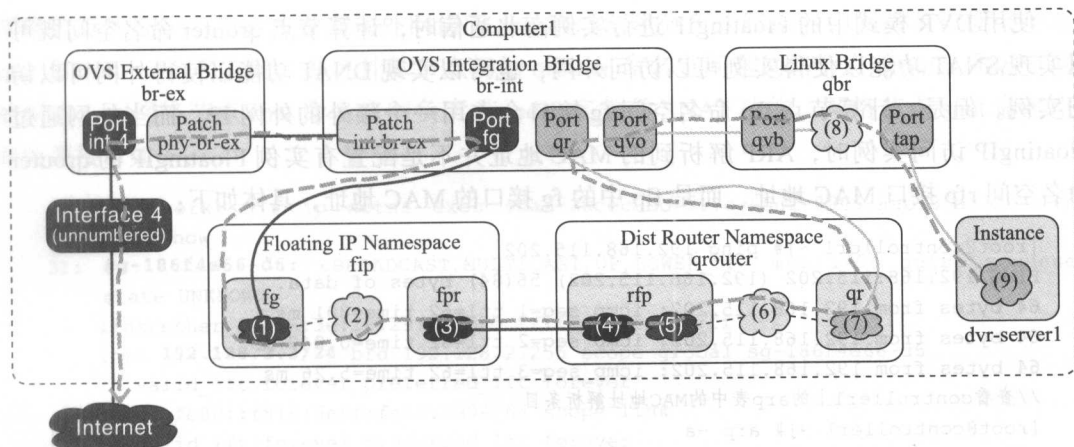


图 9-62 DVR 模式 Floating IP 实例访问外网主机

再假设外网 (192.168.115.0/24) 主机通过实例 dvr-server1 的 FloatingIP 地址 192.168.115.202 访问 compute1 上的 dvr-server1 实例 (FixedIP:192.168.1.4), 则访问数据流如图 9-63 所示。外网数据包首先进入 compute1 外网网桥 br-ex, br-ex 将数据转发到 fip 命名空间 fg 端口。fip 命名空间通过内部接口 fpr (IP 地址 169.254.31.29) 将数据包转发到 qrouter 命名空间内部接口 rfp (IP 地址 169.254.31.28)。注意 qrouter 命名空间 rfp 除了默认内部地址 (169.254.31.28) 外, 还有实例 FloatingIP 地址 (192.168.115.202)。qrouter 命名空间的 iptables 服务使用数据包中的目标地址 (即 dvr-server1 地址 192.168.1.4) 作为目的地址进行 DNAT 转换, 之后数据由 qrouter 命名空间 qr 接口 (admin-subnet1 网关接口, IP 地址 192.168.1.1) 进入集成网桥 br-int, br-int 在转发到 LinuxBridge 网桥 qbr, qbr 进行安全处理后直接转发到 dvr-server1 实例中。

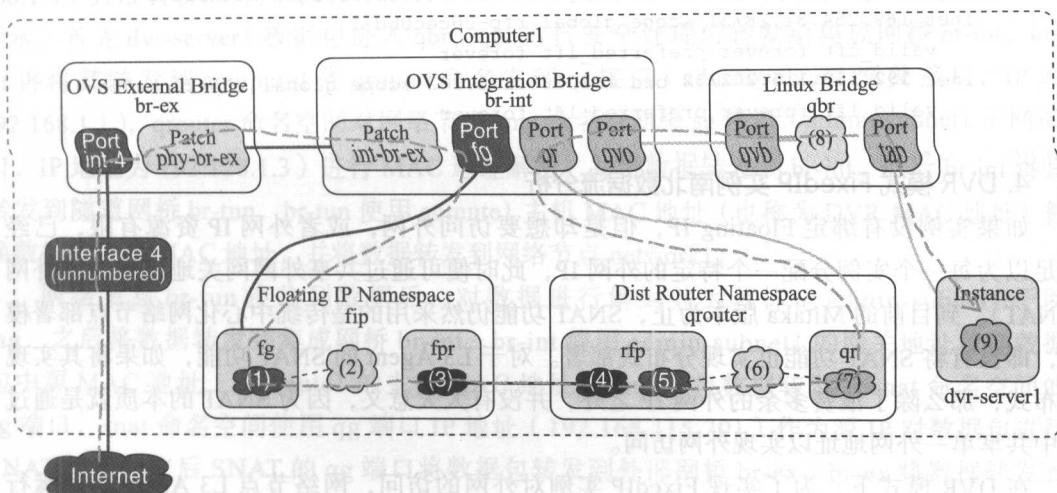


图 9-63 DVR 模式外网访问 FloatingIP 实例

使用 DVR 模式中的 FloatingIP 进行实例南北通信时, 计算节点 qrouter 命名空间既可以实现 SNAT 功能以使得实例可以访问外网, 也可以实现 DNAT 功能以使得外网可以访问实例。但是, 计算节点 fip 命名空间 fg 接口会占用一个额外的外网 IP, 而当外网通过 FloatingIP 访问实例时, ARP 解析到的 MAC 地址并不是配置有实例 FloatingIP 的 qrouter 命名空间 rfp 接口 MAC 地址, 而是 fip 中的 fg 接口的 MAC 地址, 具体如下:

```
[root@controller1 ~]# ping 192.168.115.202
PING 192.168.115.202 (192.168.115.202) 56(84) bytes of data.
64 bytes from 192.168.115.202: icmp_seq=1 ttl=62 time=801 ms
64 bytes from 192.168.115.202: icmp_seq=2 ttl=62 time=0.849 ms
64 bytes from 192.168.115.202: icmp_seq=3 ttl=62 time=5.26 ms
//查看controller1上的arp表中的MAC地址解析条目
[root@controller1 ~]# arp -a
computel (192.168.142.44) at 00:0c:29:09:bd:29 [ether] on eth1
? (192.168.115.100) at 00:50:56:e2:1c:f5 [ether] on eth0
? (192.168.115.202) at fa:16:3e:c0:54:50 [ether] on eth2
network1 (192.168.142.42) at 00:0c:29:b5:7e:68 [ether] on eth1
//computel中fip的fg端口MAC地址
[root@computel ~]# ip netns exec fip-b59c846a-6959-48df-ba81-5f0e48af69b5 ip addr
17: fg-elb88302-1d: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UNKNOWN
link/ether fa:16:3e:c0:54:50 brd ff:ff:ff:ff:ff:ff
inet 192.168.115.204/24 brd 192.168.115.255 scope global fg-elb88302-1d
valid_lft forever preferred_lft forever
//computel中qrouter命名空间rfp端口MAC地址
[root@computel ~]# ip netns exec qrouter-20cec0bd-117e-45d2-8565-aac185a51fe0 ip
addr
2: rfp-20cec0bd-1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP qlen 1000
link/ether a2:a0:95:6d:aa:12 brd ff:ff:ff:ff:ff:ff link-netnsid 0
inet 169.254.31.28/31 scope global rfp-20cec0bd-1
valid_lft forever preferred_lft forever
inet 192.168.115.202/32 brd 192.168.115.202 scope global rfp-20cec0bd-1
valid_lft forever preferred_lft forever
```

4. DVR 模式 FixedIP 实例南北数据流分析

如果实例没有绑定 Floating IP, 但是却想要访问外网, 或者外网 IP 资源有限, 已经不足以为每一个实例分配一个特定的外网 IP, 此时便可通过共享外网网关地址来访问外网 (SNAT)。到目前的 Mitaka 版本为止, SNAT 功能仍然采用的是传统中心化网络节点部署模式, 即没有将 SNAT 功能也实现分布式部署。对于 L3 Agent 的 SNAT 功能, 如果将其实现分布式, 那么除了浪费多余的外网 IP 之外, 并没有太大意义, 因为 SNAT 的本质就是通过集中共享单一外网地址以实现外网访问。

在 DVR 模式下, 为了实现 FixedIP 实例对外网的访问, 网络节点 L3 Agent 必须运行在 DVR_SNAT 模式, 并且所有 FixedIP 实例的外网访问数据流都要汇聚到网络节点, 通过

网络节点的 SNAT 功能才能实现外网访问。在用户创建分布式路由后，网络节点便会生成 snat 和 qrouter 命名空间，并且计算节点初始 qrouter 命名空间是网络节点 qrouter 命名空间的副本，而网络节点 snat 命名空间负责 FixedIP 实例的 SNAT 功能，网络节点 snat 命名空间内部接口如下：

```
[root@network1 ~]# ip netns exec snat-20cec0bd-117e-45d2-8565-aac185a51fe0 ip
addr show
31: sg-186f4e66-d6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue
state UNKNOWN
link/ether fa:16:3e:18:29:94 brd ff:ff:ff:ff:ff:ff
inet 192.128.2.3/24 brd 192.128.2.255 scope global sg-186f4e66-d6
    valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:fe18:2994/64 scope link
    valid_lft forever preferred_lft forever
32: sg-4f0c1a64-f1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue
state UNKNOWN
link/ether fa:16:3e:95:99:a3 brd ff:ff:ff:ff:ff:ff
inet 192.128.1.3/24 brd 192.128.1.255 scope global sg-4f0c1a64-f1
    valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:fe95:99a3/64 scope link
    valid_lft forever preferred_lft forever
33: qg-c52ed8b0-c5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UNKNOWN
link/ether fa:16:3e:b6:bf:35 brd ff:ff:ff:ff:ff:ff
inet 192.168.115.201/24 brd 192.168.115.255 scope global qg-c52ed8b0-c5
    valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:feb6:bf35/64 scope link
    valid_lft forever preferred_lft forever
```

现假设 compute1 计算节点的 dvr-server1 未配置 FloatingIP，仅有 FixedIP（192.168.1.4）且准备访问外网（192.168.115.0/24），则 dvr-server1 的外网访问数据流如图 9-64 所示。首先 dvr-server1 数据包进入 qbr，qbr 进行安全处理后转发给集成网桥 br-int，br-int 再将其转发到 compute1 的 qrouter 命名空间 qr 接口（admin-subnet1 网关接口，IP 为 192.168.1.1），qrouter 命名空间对网络节点 snat 命名空间中 sg 接口（admin-subnet1 子网接口，IP 地址为 192.128.1.3）进行 MAC 地址解析，并将数据转发到 br-int。然后 br-int 将其转发到隧道网桥 br-tun，br-tun 使用 compute1 主机 MAC 地址（也称为 DVR MAC 地址）替换数据包中源 MAC 地址，并将数据转发到网络节点 network1。

网络节点 br-tun 接收到数据后，对数据进行解封并为其标记 admin-subnet1 子网 tag，之后将数据转发到集成网桥 br-int，br-int 使用 admin-subnet1 的网关地址替换数据包中源 MAC 地址（compute1 的主机 MAC 地址）。br-int 将数据转发到 snat 命名空间的 sg 端口，snat 命名空间使用 qg 端口 IP 地址（192.168.115.201）作为源 IP 对数据包进行 SNAT 转换，之后 SNAT 的 qg 端口将数据包转发到外部网桥 br-ex，br-ex 将数据转发到外网。

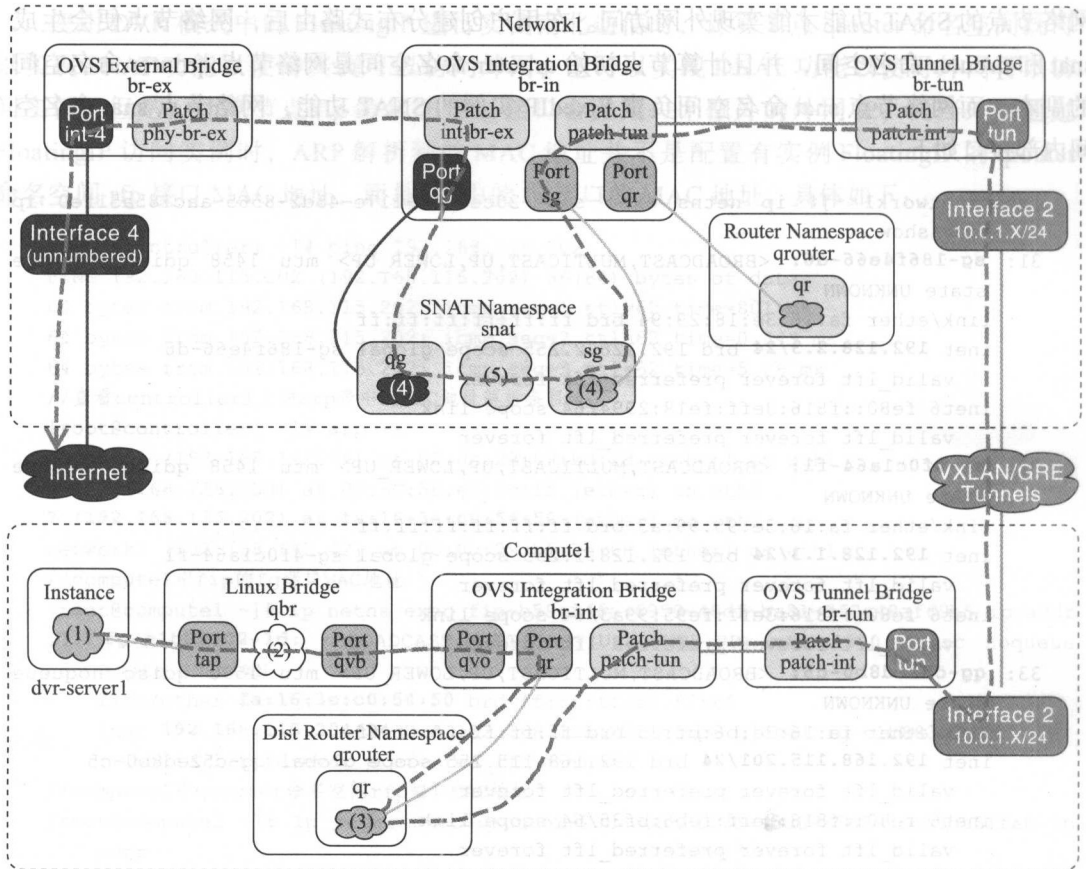


图 9-64 DVR 模式 FixedIP 实例南北通信

9.7.3 DVR 与 L3 HA 对比

DVR 是一种分布式虚拟路由技术，其最大的优势是将 L3 Agent 分布到计算节点，从而将租户子网之间的东西网络流量和南北网络流量分流到计算节点。在 DVR 中，如果租户未给实例分配并绑定 Floating IP，则租户实例与外网的通信仍然经过网络节点的 SNAT 功能来实现；如果租户为实例指定了 FloatingIP，则租户实例的东西和内部网络通信都可以通过计算节点 L3 Agent 创建的虚拟路由来实现。因此，从高可用角度而言，DVR 只是实现了 L3 Agent 的分布式高可用，而网络节点上的 SNAT 仍然采用的是传统集中化的部署方式，即网络节点 SNAT 功能的单点故障和瓶颈并没有得到解决。

L3 HA 实现了 L3 Router 全部 Layer-3 Service 功能（SNAT 和 DNAT）的高可用，但是 L3 HA 并未实现 L3 Agent 的分布式高可用，而是将 L3 Agent 部署到多个网络节点上以 A/A 或者 A/P 模式运行，通常基于 VRRP 协议的 L3 HA 主要采用 A/P 模式运行 L3 Agent 服务。因此，在某一时刻租户网络中的实例数据将全部汇聚到状态为 Active 的 L3 Agent 上，从而

造成整个集群的网络瓶颈 (BottleNeck), 即 L3 HA 部署方式只解决了 L3 Agent 服务的高可用性, 并未真正实现 L3 Agent 的分布式和扩展性, 这在大规模 OpenStack 部署环境中必然造成网络瓶颈从而限制集群的进一步扩展。

DVR 和 L3 HA 两种部署模式都在一定程度上实现了网络高可用性, 但是又各自存在一些缺陷。如 DVR 实现了 L3 Agent 的分布式, 解决了网络瓶颈和集群扩展性问题, 但是留下了网络节点 SNAT 的单点故障问题; 而 L3 HA 解决了 SNAT 和 DNAT 的单点故障问题, 但是留下了网络节点性能瓶颈和扩展性问题。因此, 对于一个规模化的 OpenStack 集群, 最理想的 Neutron 服务高可用部署模式应该是二者的结合, 即 DVR 模式与 L3 SNAT 高可用部署模式同时启用, 这样不仅利用 L3 Agent 的分布式解决了网络性能问题和扩展性问题, 同时也解决了网络节点 SNAT 的单点故障问题。OpenStack 社区在 Liberty 版本中对该部署模式进行了讨论和开发, 并且在 Liberty 版本已经可以实现此类高可用部署, 但是直到 Mitaka 版本正式发布, 社区才宣布 Neutron 支持 SNAT HA 与 DVR 兼容部署 (<http://docs.openstack.org/releasenotes/Neutron/mitaka.html>)。下一节将重点讲解 Mitaka 版本中网络节点 SNAT HA 与 DVR 兼容部署的实现。

9.8 DVR/L3 HA 高可用方案

9.8.1 DVR/L3 HA 高可用部署实现

分布式虚拟路由 DVR 功能在 Juno 版本发行, 但是在 Mitaka 版本之前, 社区一直没有正式宣称 DVR 与 L3 HA 可以兼容部署, 尽管在 Kilo 版本中便有关于 DVR 与 L3 HA 兼容部署的方案讨论, 直到 Liberty 版本后, 社区才真正开始开发验证并实现二者结合的高可用网络部署, 并且在 Mitaka 版本发行时, 社区正式宣布 Neutron8.0 支持使用 L3 HA 增强实现 DVR。为了区别 L3 HA 高可用部署方案和 DVR 分布式路由方案, 通常将其称为 DVR/L3 HA 高可用部署方案。DVR/L3 HA 高可用部署模式不仅解决了网络瓶颈带来的性能问题, 同时也解决了网络节点 SNAT 高可用的问题。DVR/L3 HA 可以简单看成是对 DVR 的功能扩展, 即在保持 DVR 功能不变的情况下, 将网络节点的 L3 SNAT 功能通过 L3 HA 实现高可用, 或者也可以认为 DVR/L3 HA 是对 DVR 模式下网络节点 SNAT 单点故障解决后的补充方案。在用户同时启用 DVR 和 L3 HA 功能后, 租户便可同时使用 `--distributed` 和 `--ha` 选项进行路由创建。DVR/L3 HA 模式下, Neutron L3 服务在各个节点的部署和运行模式如图 9-65 所示。

DVR/L3 HA 高可用网络架构各个节点运行网络服务与 DVR 模式类似, 不同之处在于相对 DVR 需要增加冗余网络节点, 网络节点需要运行 Keepalived 进程以实现 SNAT 高可用。而 L3 Agent 在网络节点和计算节点上的运行模式均未变, 即网络节点以 `dvr_snat` 模式运行, 计算节点以 `dvr` 模式运行。DVR/L3 HA 模式节点服务如图 9-66 所示。

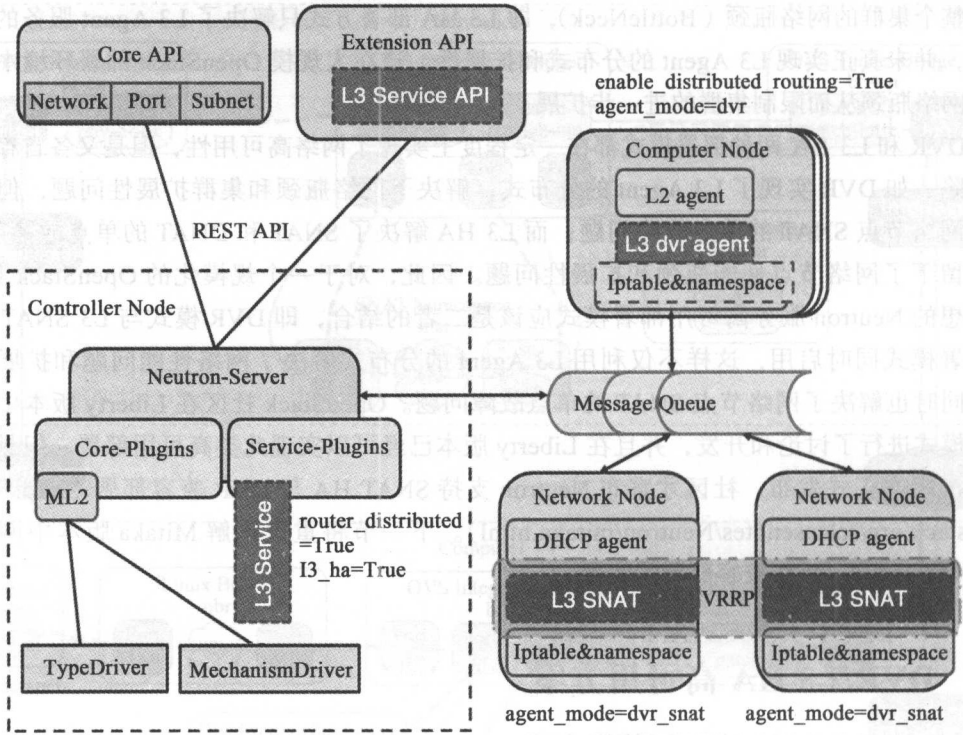


图 9-65 DVR/L3 HA 高可用中 L3 Service 部署及运行模式

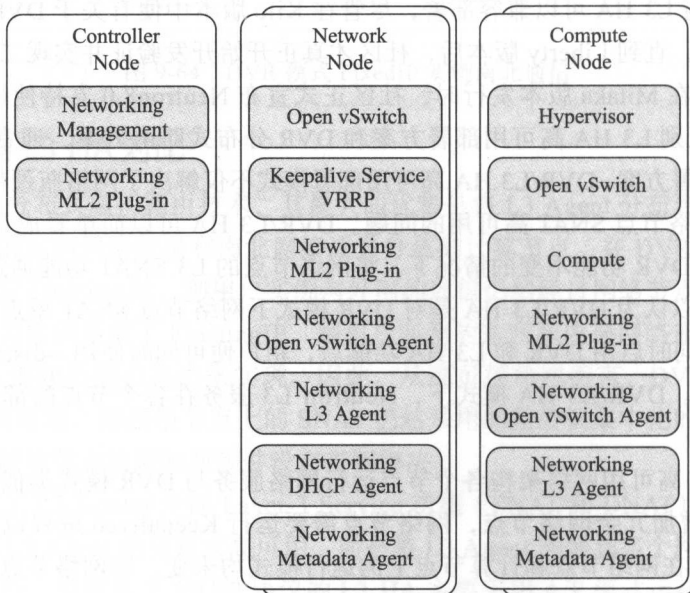


图 9-66 DVR/L3 HA 节点网络服务布局

DVR/L3 HA 高可用网络部署与 DVR 网络架构的部署实现类似, 所不同之处在于 DVR/L3 HA 需要配置额外的网络节点以实现 SNAT 高可用, 并在控制节点同时启用 DVR 和 L3 HA 功能。而 DVR/L3 HA 与传统 L3 HA 高可用部署的不同之处在于, 传统 L3 HA 实现的是网络节点 `qrouter` 命名空间的高可用, 但是 DVR/L3 HA 实现的是网络节点 `snat` 命名空间的高可用, 目前为止仍然不能在 DVR 模式下实现网络节点 `qrouter` 命名空间的高可用。本节以五节点 OpenStack 集群 DVR/L3 HA 高可用网络部署为例, 其中网络节点两个, 计算节点两个, 控制节点一个, 各个节点运行的网络服务如图 9-66 所示。DVR/L3 HA 高可用网络具体部署过程分为控制节点部署、计算节点部署和网络节点部署。

1. 控制节点配置

1) /etc/neutron/neutron.conf 文件配置。

```
.....
[DEFAULT]
core_plugin = ml2
service_plugins = router
allow_overlapping_ips = True
//启用分布式路由DVR功能
router_distributed = True
//禁用L3 HA功能, 在Mitaka版本后才可以同时启用DVR与L3 HA功能
l3_ha=True
// L3 HA心跳默认网络
l3_ha_net_cidr = 169.254.192.0/18
//L3 HA 主备Router最大个数, 即全部Master/Backup路由个数
max_l3_agents_per_router = 4
// L3 HA Router创建必要条件, 即L3 agent正常运行的最小数目
min_l3_agents_per_router = 2
//dhcp高可用配置, 设置为集群中运行的DHCP Agent的数目, 通常为网络节点数目
dhcp_agents_per_network =2
.....
```

2) /etc/neutron/plugins/ml2/ml2_conf.ini 文件配置。

```
.....
//////////ML2驱动和租户网络类型配置//////////
[ml2]
//ML2插件TypeDriver列表
type_drivers = flat,vlan,gre,vxlan
//ML2插件MechanismDriver列表, 这里选择的是openvswitch和l2population, l2_population仅
对类型为
//GRE/VxLAN的租户网络有效
mechanism_drivers = openvswitch,l2population
//租户网络(Project网络)类型, 第一个值将是常规租户创建网络时的默认值, 这里常规租户创建的将是
VLAN网络,
tenant_network_types = vlan,gre,vxlan
extension_drivers = port_security
//////////
```



```

////////////////////网络mapping和ID范围配置////////////////////
//Flat类型网络创建需要指定物理网络名称(--provider:physical_network)，其值在此设置
[ml2_type_flat]
flat_networks = external
//VLAN类型网络创建需要指定物理网络名称和VLAN ID，此处设置了external和vlan两个物理网络
//物理网络名称后面可以指定该网络允许的VLAN ID，不设置VLAN ID表示ID范围不受限
[ml2_type_vlan]
network_vlan_ranges = external,vlan:1:1024
//GRE类型网络创建不需要物理网络名称，此处只需设置一个GRE隧道ID范围即可
[ml2_type_gre]
tunnel_id_ranges = 1:1000
//VxLAN网络创建不需要物理网络名称，此处只需设置一个VxLAN隧道ID范围即可
[ml2_type_vxlan]
vni_ranges = 1:1000
////////////////////
[securitygroup]
firewall_driver = iptables_hybrid
.....

```

3) 启动 Neutron-Server 服务。

```

systemctl start neutron-server.service
systemctl enable neutron-server.service

```

2. 网络节点配置

1) /etc/neutron/plugins/ml2/openvswitch_agent.ini 文件配置。

```

.....
[ovs]
//指定用于GRE/VxLAN的隧道IP接口地址(使用具体IP地址替换TUNNEL_INTERFACE_IP_ADDRESS)
local_ip = TUNNEL_INTERFACE_IP_ADDRESS
//设置对应物理网络的网桥，此处的vlan和external两个物理网络名称必须与ml2_conf.ini文件中设定
值一致
bridge_mappings = vlan:br-vlan,external:br-ex
[agent]
tunnel_types = gre,vxlan
//l2_population仅对GRE/VxLAN网络有效
l2_population = True
//网络节点上启用分布式路由支持
enable_distributed_routing = True
arp_responder = True

[securitygroup]
firewall_driver = iptables_hybrid
.....

```

2) /etc/neutron/l3_agent.ini 文件配置。

```

.....
[DEFAULT]
interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver

```

```
external_network_bridge =
//网络节点L3 Agent工作模式为dvr_snat
agent_mode = dvr_snat
router_delete_namespaces = True
// vrrp协议广播心跳时间间隔
ha_vrrp_advert_int = 2
//Keepalived ha配置文件路径
ha_confs_path = /opt/stack/data/Neutron/ha_confs
.....
```

3) /etc/neutron/dhcp_agent.ini 文件配置。

```
.....
[DEFAULT]
interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver
enable_isolated_metadata = True
.....
```

4) /etc/neutron/metadata_agent.ini 文件配置。

```
.....
[DEFAULT]
nova_metadata_ip = controller
metadata_proxy_shared_secret = METADATA_SECRET
.....
```

5) 启动网络节点服务。

```
systemctl start openvswitch.service
systemctl enable openvswitch.service
systemctl start neutron-l3-agent.service
systemctl enable neutron-l3-agent.service
systemctl start dhcp-l3-agent.service
systemctl enable dhcp-l3-agent.service
systemctl start openvswitch-l3-agent.service
systemctl enable openvswitch-l3-agent.service
systemctl start metadata-l3-agent.service
systemctl enable metadata-l3-agent.service
```

3. 计算节点配置

1) /etc/neutron/plugins/ml2/openvswitch_agent.ini 文件配置。

```
.....
[ovs]
//使用具体的隧道IP地址替换TUNNEL_INTERFACE_IP_ADDRESS
local_ip = TUNNEL_INTERFACE_IP_ADDRESS
bridge_mappings = vlan:br-vlan,external:br-ex
[agent]
tunnel_types = gre,vxlan
enable_distributed_routing = True
l2_population = True
arp_responder = True
```

```
[securitygroup]
firewall_driver = iptables_hybrid
.....
```

2) /etc/neutron/l3_agent.ini 文件配置。

```
.....
[DEFAULT]
interface_driver = Neutron.agent.linux.interface.OVSInterfaceDriver
external_network_bridge =
//配置计算节点L3 agent为dvr模式
agent_mode = dvr
.....
```

3) /etc/neutron/plugins/ml2/metadata.ini 文件配置。

```
.....
[DEFAULT]
nova_metadata_ip = controller
metadata_proxy_shared_secret = METADATA_SECRET
.....
```

4) 启动计算节点网络服务。

```
systemctl start openvswitch.service
systemctl enable openvswitch.service
systemctl start neutron-openvswitch-agent.service
systemctl enable neutron-openvswitch-agent.service
systemctl start neutron-l3-agent.service
systemctl enable neutron-l3-agent.service
systemctl start metadata-l3-agent.service
systemctl enable metadata-l3-agent.service
```

各个节点网络服务配置并启动完成之后，在控制节点加载管理员权限，验证 DVR/L3 HA 模式下各个节点 Neutron 代理服务启动情况。正常情况下，各个节点网络代理服务运行情况应该如下：

```
[root@controller1 ~]# neutron agent-list
+-----+-----+-----+-----+-----+-----+
| id | agent_type | host | alive | admin_state_up | binary |
+-----+-----+-----+-----+-----+-----+
| ...6bea | Metadata agent | compute1 | :- ) | True | neutron-metadata-agent |
| ...7529 | Metadata agent | compute2 | :- ) | True | neutron-metadata-agent |
| ...a526 | Open vSwitch agent | network1 | :- ) | True | neutron-openvswitch-agent |
| ...b275 | Open vSwitch agent | network2 | :- ) | True | neutron-openvswitch-agent |
```

...84b2 dhcp-agent	DHCP agent	network2	:-)	True	neutron-
...91aa dhcp-agent	DHCP agent	network1	:-)	True	neutron-
...3e5a l3-agent	L3 agent	network2	:-)	True	neutron-
...4666 l3-agent	L3 agent	network1	:-)	True	neutron-
...30a9 metadata-agent	Metadata agent	network1	:-)	True	neutron-
...cf49 openvswitch-agent	Open vSwitch agent	compute2	:-)	True	neutron-
...2cf1 l3-agent	L3 agent	compute1	:-)	True	neutron-
...247e l3-agent	L3 agent	compute2	:-)	True	neutron-
...1b7f openvswitch-agent	Open vSwitch agent	compute1	:-)	True	neutron-
...c5a7 metadata-agent	Metadata agent	network2	:-)	True	neutron-
+-----+-----+-----+-----+-----+					
-----+					

正常情况下，L3 Agent、OpenvSwitch Agent 和 Metadata Agent 运行在两个计算节点和两个网络节点上，DHCP Agent 运行在两个网络节点上，并且计算节点上的 L3 Agent 运行在 DVR 模式，而网络节点的 L3 Agent 运行在 DVR_SNAT 模式。在确认各个网络服务运行正常之后，用户便可创建分布式高可用路由（DVR HA Router）。

9.8.2 DVR/L3HA 高可用验证与分析

1. 网络与 DVR HA Router 创建

1) 首先创建 External 网络 ext-net 及其子网 ext-subnet。

本节将创建 Flat 类型的 External 网络和 GRE 类型的 Project 网络，用户也可以选择创建 VLAN 类型的 External 网络和 VxLAN 或 VLAN 类型的 Project 网络。ext-subnet 子网可用 IP 地址段为 192.168.115.200~192.168.115.250，外部网络禁止使用 DHCP 功能。

```
//创建外网ext-net
[root@controller1 ~]# neutron net-create ext-net --router:external True \
--provider:physical_network external--provider:network_type flat
//创建外网子网ext-subnet
[root@controller1 ~]# neutron subnet-create ext-net 192.168.115.0/24 --name ext-subnet
--allocation-pool start=192.168.115.200,end=192.168.115.250 --disable-dhcp
--gateway
192.168.115.254
[root@controller1 ~]# neutron net-list
+-----+-----+-----+-----+-----+
|      id      | name      | subnets |
```

```
+-----+
|...48af69b5 | ext-net | 1be010f3-c5f2-42a4-87db-ccd64c56b70e 192.168.115.0/24|
+-----+
```

2) 创建 Project 网络 admin-net 及其子网 admin-subnet。

admin-subnet 为租户子网，租户可以自定义子网名称及其 IP 地址范围，每个租户可以创建多个租户子网，为了便于后续网络分析，此处创建两个不同的租户网络及其子网：admin-net1 及其子网 admin-subnet1，admin-net2 及其子网 admin-subnet2。注意根据 tenant_networks_type 配置参数，仅有管理员才可以创建 GRE/VxLAN 类型的 Project 网络，普通租户默认创建的是 VLAN 类型租户网络。

```
[root@controller1 ~]# neutron net-create admin-net1 --provider:network_type gre
[root@controller1 ~]# neutron net-create admin-net2 --provider:network_type gre
[root@controller1 ~]# neutron subnet-create admin-net1 --name admin-subnet1
--gateway 192.128.1.1 192.128.1.0/24
[root@controller1 ~]# neutron subnet-create admin-net2 --name admin-subnet2
--gateway 192.128.2.1 192.128.2.0/24
[root@controller1 ~]# neutron net-list
```

```
+-----+
| id | name | subnets |
+-----+
|...69b5 | ext-net | 1be010f3-c5f2-42a4-87db-ccd64c56b70e 192.168.115.0/24 |
|...d44d | admin-net1 | ec673050-4d9c-4574-bda2-fb954e280dd7 192.128.1.0/24 |
|...f192 | admin-net2 | 88f9c09c-5fc9-4468-87da-241864e1e3b8 192.128.2.0/24 |
+-----+
```

```
[root@controller1 ~]# neutron subnet-list
```

```
+-----+
| id | name | cidr | allocation_pools |
+-----+
|...b70e | ext-subnet | 192.168.115.0/24 | {"start": "192.168.115.200", "end": "192.168.115.250"} |
|...e3b8 | admin-subnet1 | 192.128.1.0/24 | {"start": "192.128.1.2", "end": "192.128.1.254"} |
|...0dd7 | admin-subnet2 | 192.128.2.0/24 | {"start": "192.128.2.2", "end": "192.128.2.254"} |
+-----+
```

3) 创建分布式高可用路由 dvr_ha_router。

```
[root@controller1 ~]# neutron router-create dvr_ha_router
Created a new router:
```

```
+-----+
| Field | Value |
+-----+
| admin_state_up | True |
| availability_zone_hints | |
+-----+
```



```
| availability_zones | |
| description | |
| distributed | True |
| external_gateway_info | |
| ha | True |
| id | 98d44469-cbf8-4be0-8d2d-b764148b19cb |
| name | dvr_ha_router |
| routes | |
| status | ACTIVE |
| tenant_id | cbe811690f3c432aa59fbedcf918a793 |
```

```
[root@controller1 ~]# neutron router-list
```

```
+-----+-----+-----+
| id | name | external_gateway_info |
distributed | ha |
+-----+-----+-----+
| 98d44469-cbf8-4be0-8d2d-b764148b19cb | dvr_ha_router | null |
True | True |
```

```
//新增HA网络
```

```
[root@controller1 ~]# neutron subnet-list
```

```
+-----+-----+-----+
| id | name | cidr | allocation_pools |
+-----+-----+-----+
| ...b70e | ext-subnet | 192.168.115.0/24 | {"start": "192.168.115.200", "end": "192.168.115.250"} |
| ...5a04 | HA subnet tenant ...a793 | 169.254.192.0/18 | {"start": "169.254.192.1", "end": "169.254.255.254"} |
| ...e3b8 | admin-subnet2 | 192.128.2.0/24 | {"start": "192.128.2.2", "end": "192.128.2.254"} |
| ...0dd7 | admin-subnet1 | 192.128.1.0/24 | {"start": "192.128.1.2", "end": "192.128.1.254"} |
```

4) 检查计算节点和网络节点网络命名空间。

```
//网络节点network1命名空间
```

```
[root@network1 ~]# ip netns
```

```
qrouter-98d44469-cbf8-4be0-8d2d-b764148b19cb
```

```
snat-98d44469-cbf8-4be0-8d2d-b764148b19cb
```

```
qdhcp-67fb0d01-28fd-4cb4-871b-c1517561f192
```

```
//网络节点network2命名空间
```

```
[root@network2 ~]# ip netns
```

```
qrouter-98d44469-cbf8-4be0-8d2d-b764148b19cb
```

```

snat-98d44469-cbf8-4be0-8d2d-b764148b19cb
qdhcp-67fb0d01-28fd-4cb4-871b-c1517561f192
//计算节点compute1命名空间
[root@compute1 ~]# ip netns
//计算节点compute2命名空间
[root@compute2 ~]# ip netns

```

5) 查看网络节点 qrouter 和 snat 命名空间接口配置情况。

```

//network2网络节点snat命名空间
[root@network2 ~]# ip netns exec snat-98d44469-cbf8-4be0-8d2d-b764148b19cb ip
addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
11: ha-7783f8a4-4c: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue
    state UNKNOWN
    link/ether fa:16:3e:81:dc:87 brd ff:ff:ff:ff:ff:ff
    inet 169.254.192.1/18 brd 169.254.255.255 scope global ha-7783f8a4-4c
        valid_lft forever preferred_lft forever
    inet 169.254.0.1/24 scope global ha-7783f8a4-4c
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe81:dc87/64 scope link
        valid_lft forever preferred_lft forever
//network2网络节点qrouter命名空间
[root@network2 ~]# ip netns exec qrouter-98d44469-cbf8-4be0-8d2d-b764148b19cb ip
addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
//network1网络节点snat命名空间
[root@network1 ~]# ip netns exec snat-98d44469-cbf8-4be0-8d2d-b764148b19cb ip
addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
36: ha-1fb36e5d-13: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1458 qdisc noqueue
    state UNKNOWN
    link/ether fa:16:3e:9c:df:c8 brd ff:ff:ff:ff:ff:ff
    inet 169.254.192.2/18 brd 169.254.255.255 scope global ha-1fb36e5d-13
        valid_lft forever preferred_lft forever
    inet 169.254.0.1/24 scope global ha-1fb36e5d-13

```

```

    valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe9c:dfc8/64 scope link
    valid_lft forever preferred_lft forever
//network1网络节点qrouter命名空间
[root@network1 ~]# ip netns exec qrouter-98d44469-cbf8-4be0-8d2d-b764148b19cb ip
addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever

```

6) 检查分布式高可用路由 dvr_ha_router 的主从关系。

```

[root@controller1 ~]# neutron l3-agent-list-hosting-router dvr_ha_router
+-----+-----+-----+-----+-----+
| id                  | host      | admin_state_up | alive | ha_state |
+-----+-----+-----+-----+-----+
| 71a2efed-ee93-4721-9372-51bd099f3e5a | network2 | True           | :- ) | active   |
| 7509137d-eda6-471c-84df-76fc15814666 | network1 | True           | :- ) | standby  |
+-----+-----+-----+-----+-----+

```

7) 为分布式高可用路由 dvr_ha_router 设置外网网关和租户子网接口。

```

[root@controller1 ~]# neutron router-interface-add dvr_ha_router admin-subnet1
Added interface e22273c7-93e7-4897-8396-384bd5d04dd4 to router dvr_ha_router.
[root@controller1 ~]# neutron router-interface-add dvr_ha_router admin-subnet2
Added interface 3e9dccf8-31a2-4faa-9edc-92ac5febb932 to router dvr_ha_router.
[root@controller1 ~]# neutron router-gateway-set dvr_ha_router ext-net
Set gateway for router dvr_ha_router
[root@controller1 ~]# neutron router-list
+-----+-----+-----+-----+-----+
| id          | name          | external_gateway_info |
|distributed| ha            |
+-----+-----+-----+-----+-----+
| ...19cb     | dvr_ha_router | {"network_id": "b59c846a-6959-48df-ba81-5f0e48af69b5", |
| True        | True          | "enable_snat": true, "external_fixed_ips": [{"subnet_id": |
|              |               | "1be010f3-c5f2-42a4-87db-ccd64c56b70e", "ip_address": |
|              |               | "192.168.115.206"}]} |
|              |               |
+-----+-----+-----+-----+-----+

```

从上述结果可以看出，分布式高可用路由 dvr_ha_router 目前只具备 HA 高可用性，并且 MasterRouter 位于 network1 网络节点，但是计算节点上没有任何网络命名空间，即 dvr_

ha_router 仍然不具备分布式。需要指出的是,网络节点 qrouter 命名空间中并没有创建 HA 网络,只在 snat 命名空间中创建了 HA 网络,这与 L3 HA 网络架构是不同的。L3 HA 的网络架构下 HA 网络位于 qrouter 命名空间中,针对的是 qrouter 命名空间的高可用,而 DVR/SNAT HA 针对的是网络节点 snat 命名空间的高可用。因此,一定要区分清楚,目前部署实现的是 DVR/L3 SNAT HA 网络,而不是 DVR/L3 Router HA 网络,后者在 Mitaka 版本中仍然不被支持,即目前仍然不能实现 DVR 模式下网络节点 qrouter 命名空间的高可用。此外,只有在用户创建实例时,计算节点 L3 Agent 才会在本节点创建分布式路由,下面在 admin-net1 和 admin-net2 租户网络中创建两个实例 dvr_ha_server1 和 dvr_ha_server2。

2. 创建租户实例并验证计算节点命名空间

1) 创建租户实例 dvr_ha_server1 和 dvr_ha_server2。

```
[root@controller1 ~]# nova boot --flavor 1 --image cirros-0.3.4-x86_64 --key-
name admin-key --security-group default --nic net-id=748d632a-3235-4313-87f8-
f6745e0c5416 dvr_ha_server1
[root@controller1 ~]# nova boot --flavor 1 --image cirros-0.3.4-x86_64 --key-
name admin-key --security-group default --nic net-id=262e09f3-ae8d-48c7-bba3-
ff62f15da77a dvr_ha_server2
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State	Networks
...cac5	dvr_ha_server1	ACTIVE	-	Running	admin-net1=192.128.1.5
...d944	dvr_ha_server2	ACTIVE	-	Running	admin-net2=192.128.2.5

```
// dvr_ha_server1位于compute1
[root@controller1 ~]# nova hypervisor-servers compute1
```

ID	Name	Hypervisor ID	Hypervisor Hostname
....-091965f1cac5	instance-00000027	1	compute1

```
//dvr_ha_server2位于compute2
[root@controller1 ~]# nova hypervisor-servers compute2
```

ID	Name	Hypervisor ID	Hypervisor Hostname
....-fc4f5a5bd944	instance-00000028	2	compute2

2) 检查分布式高可用路由主备状态。

```
[root@controller1 ~]# neutron l3-agent-list-hosting-router dvr_ha_router
```

id	host	admin_state_up	alive	ha_state
----	------	----------------	-------	----------

```
| 7509137d-eda6-471c-84df-76fc15814666 | network1 | True | :- ) | standby |
| c4258f65-1cec-4916-8d01-b84d3bea247e | compute2 | True | :- ) | standby |
| 71a2efed-ee93-4721-9372-51bd099f3e5a | network2 | True | :- ) | active |
| b8e0fc29-e196-4b4a-8db7-842a4bbb2cf1 | compute1 | True | :- ) | standby |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

3) 为实例 dvr_ha_server1 和 dvr_ha_server2 绑定 FloatingIP。

```
[root@controller1 ~]# nova floating-ip-associate dvr_ha_server1 192.168.115.207
[root@controller1 ~]# nova floating-ip-associate dvr_ha_server2 192.168.115.208
[root@controller1 ~]# nova list
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ID | Name | Status | Task State | Power State | Networks |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ...ac5 | dvr_ha_server1 | ACTIVE | - | Running | admin-net1=192.128.1.5, 192.168.115.207 |
| ...944 | dvr_ha_server2 | ACTIVE | - | Running | admin-net2=192.128.2.5, 192.168.115.208 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

4) 检查网络节点和计算节点命名空间。

```
//compute1计算节点命名空间
[root@compute1 ~]# ip netns
fip-b59c846a-6959-48df-ba81-5f0e48af69b5
qrouter-98d44469-cbf8-4be0-8d2d-b764148b19cb
//compute2计算节点命名空间
[root@compute2 ~]# ip netns
fip-b59c846a-6959-48df-ba81-5f0e48af69b5
qrouter-98d44469-cbf8-4be0-8d2d-b764148b19cb
//网络节点network1命名空间
[root@network1 ~]# ip netns
qrouter-98d44469-cbf8-4be0-8d2d-b764148b19cb
snat-98d44469-cbf8-4be0-8d2d-b764148b19cb
qdhcp-67fb0d01-28fd-4cb4-871b-c1517561f192
//网络节点network2命名空间
[root@network2 ~]# ip netns
qrouter-98d44469-cbf8-4be0-8d2d-b764148b19cb
snat-98d44469-cbf8-4be0-8d2d-b764148b19cb
qdhcp-67fb0d01-28fd-4cb4-871b-c1517561f192
```

在计算节点 compute1 和 compute2 上分别创建实例 dvr_ha_server1 和 dvr_ha_server2 后, 计算节点 L3 Agent 在本地创建了 qrouter 命名空间, 计算节点命名空间是网络节点 qrouter 命名空间的副本。这里需要注意的是 DVR/L3 HA 只实现 snat 命名空间的高可用, 因此虽然 l3-agent-list-hosting-router 命令显示计算节点为高可用 Router 的 Standby

节点，但是由于计算节点没有 snat 命名空间，因此高可用 snat 只会在网络节点之间切换。此外，当用户为实例绑定 Floating IP 后，L3 Agent 又在计算节点上创建了 fip 命名空间，fip 命名空间主要用于实例南北网络数据通信。实例创建之后的网络拓扑如图 9-67 所示。

3. DVR/L3 HA 高可用功能验证与分析

1) 分布式高可用路由 FixedIP 实例的 SNAT 高可用验证。

为了验证分布式高可用路由的 SNAT 高可用性，先取消租户实例 dvr_ha_server1 和 dvr_ha_server2 的 FloatingIP，使其仅有 FixedIP，并从 dvr_ha_server1 和 dvr_ha_server2 访问外部网络（如 Internet）。在确保实例正常访问外网的情况下，关闭 SNAT 的 Master 网络节点，观察 SNAT 是否切换到另外网络节点，并验证外网访问是否正常。当前 dvr_ha_server1 和 dvr_ha_server2 实例状态如下：

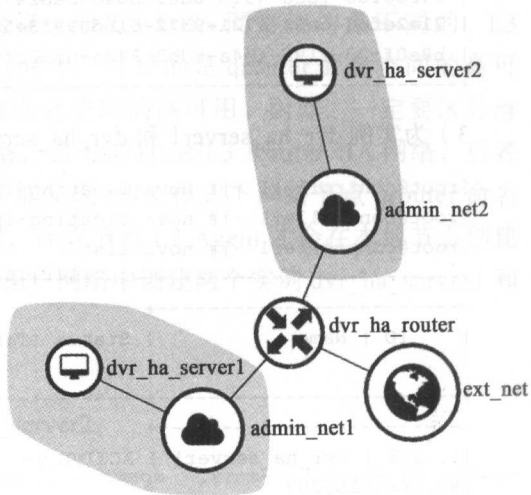


图 9-67 租户实例网络拓扑

```
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State	Networks
...1965f1cac5	dvr_ha_server1	ACTIVE	-	Running	admin-net1=192.128.1.5
...4f5a5bd944	dvr_ha_server2	ACTIVE	-	Running	admin-net2=192.128.2.5

分布式高可用路由状态如下：

```
[root@controller1 ~]# neutron l3-agent-list-hosting-router dvr_ha_router
```

id	host	admin_state_up	alive	ha_state
7509137d-eda6-471c-84df-76fc15814666	network1	True	:-)	active
c4258f65-1cec-4916-8d01-b84d3bea247e	compute2	True	:-)	standby
71a2efed-ee93-4721-9372-51bd099f3e5a	network2	True	:-)	standby
b8e0fc29-e196-4b4a-8db7-842a4bbb2cf1	compute1	True	:-)	standby

访问租户实例 dvr_ha_server2，并从实例中 ping 任意 Internet 网址，ping 测试结果如图 9-68 所示，结果表明 dvr_ha_server2 可以正常访问外网（dvr_ha_server1 结果是一致的，不再测试）。

```
# hostname
dvr-ha-server2
# ping www.baidu.com
PING www.baidu.com (61.135.169.121): 56 data bytes
64 bytes from 61.135.169.121: seq=0 ttl=127 time=39.852 ms
64 bytes from 61.135.169.121: seq=1 ttl=127 time=43.844 ms
64 bytes from 61.135.169.121: seq=2 ttl=127 time=50.841 ms
64 bytes from 61.135.169.121: seq=3 ttl=127 time=46.676 ms
64 bytes from 61.135.169.121: seq=4 ttl=127 time=61.446 ms
64 bytes from 61.135.169.121: seq=5 ttl=127 time=43.772 ms
64 bytes from 61.135.169.121: seq=6 ttl=127 time=42.040 ms
64 bytes from 61.135.169.121: seq=7 ttl=127 time=43.352 ms
64 bytes from 61.135.169.121: seq=8 ttl=127 time=43.445 ms
64 bytes from 61.135.169.121: seq=9 ttl=127 time=42.863 ms
64 bytes from 61.135.169.121: seq=10 ttl=127 time=49.370 ms
64 bytes from 61.135.169.121: seq=11 ttl=127 time=41.667 ms
64 bytes from 61.135.169.121: seq=12 ttl=127 time=48.878 ms
64 bytes from 61.135.169.121: seq=13 ttl=127 time=51.086 ms
```

图 9-68 dvr_ha_server2 正常访问 Internet 网络

现关闭 network1 网络节点，并观察分布式高可用路由情况：

```
[root@controller1 ~]# neutron l3-agent-list-hosting-router dvr_ha_router
```

id	host	admin_state_up	alive	ha_state
7509137d-eda6-471c-84df-76fc15814666	network1	True	xxx	standby
c4258f65-1cec-4916-8d01-b84d3bea247e	compute2	True	:-)	standby
71a2efed-ee93-4721-9372-51bd099f3e5a	network2	True	:-)	active
b8e0fc29-e196-4b4a-8db7-842a4bbb2cf1	compute1	True	:-)	standby

可以看到，network1 网络节点关闭后，L3 SNAT 已经自动切换至 network2 节点。现继续测试 dvr_ha_server2 对 Internet 的访问情况，测试结果如图 9-69 所示，dvr_ha_server2 仍然可以访问外网。

```
# hostname
dvr-ha-server2
# ping www.baidu.com
PING www.baidu.com (61.135.169.125): 56 data bytes
64 bytes from 61.135.169.125: seq=0 ttl=127 time=50.895 ms
64 bytes from 61.135.169.125: seq=1 ttl=127 time=49.267 ms
64 bytes from 61.135.169.125: seq=2 ttl=127 time=48.101 ms
64 bytes from 61.135.169.125: seq=3 ttl=127 time=46.170 ms
64 bytes from 61.135.169.125: seq=4 ttl=127 time=57.297 ms
64 bytes from 61.135.169.125: seq=5 ttl=127 time=45.783 ms
64 bytes from 61.135.169.125: seq=6 ttl=127 time=44.683 ms
64 bytes from 61.135.169.125: seq=7 ttl=127 time=41.845 ms
64 bytes from 61.135.169.125: seq=8 ttl=127 time=42.531 ms
64 bytes from 61.135.169.125: seq=9 ttl=127 time=40.790 ms
64 bytes from 61.135.169.125: seq=10 ttl=127 time=48.570 ms
64 bytes from 61.135.169.125: seq=11 ttl=127 time=52.897 ms
```

图 9-69 dvr_ha_server2 正常访问 Internet

最后，将 network2 节点也关闭，即网络节点不再提供 L3 SNAT 服务，此时分布式高可用路由状态如下：

```
[root@controller1 ~]# neutron l3-agent-list-hosting-router dvr_ha_router
```

id	host	admin_state_up	alive	ha_state
----	------	----------------	-------	----------

```

| 7509137d-eda6-471c-84df-76fc15814666 | network1 | True | xxx | standby |
| c4258f65-1cec-4916-8d01-b84d3bea247e | compute2 | True | :-) | standby |
| 71a2efed-ee93-4721-9372-51bd099f3e5a | network2 | True | xxx | standby |
| b8e0fc29-e196-4b4a-8db7-842a4bbb2cf1 | compute1 | True | :-) | standby |
+-----+-----+-----+-----+

```

现在继续测试 dvr_ha_server2 对 Internet 的访问情况，正常情况下此时 dvr_ha_server2 应该无法访问 Internet 网络，测试结果如图 9-70 所示。

```

# hostname
dvr-ha-server2
# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes

--- 8.8.8.8 ping statistics ---
7 packets transmitted, 0 packets received, 100% packet loss
# _

```

图 9-70 dvr_ha_server2 无法访问 Internet

上述结果充分验证了分布式高可用路由在网络节点故障情况下的高可用性，即 DVR/L3 SNAT HA 方案解决了 DVR 模式网络节点 SNAT 功能单点故障的问题，在网络节点故障情况下，L3 SNAT 自动进行了高可用切换，并继续对外提供 SNAT 服务。而网络节点全部故障后，FixedIP 实例将无法访问外部网络，这也充分说明在 DVR 模式下实现网络节点 SNAT 高可用性具有非常重要的作用。分布式高可用路由中 FixedIP 实例对外部网络的访问数据流与 DVR 模式下完全相同，这里不再分析。

2) 分布式高可用路由 FixedIP 实例的东西网络通信验证。

根据分布式路由的设计思想，一旦路由被分布到计算节点，则租户不同子网之间的实例东西向通信将不再汇聚到网络节点，而是直接由计算节点 L3 Agent 提供路由功能。因此，在 DVR/L3 HA 高可用网络架构中，网络节点将不再影响租户实例的东西向通信。为了进行验证，现将 network1 和 network2 节点同时关闭，此时集群中不再存在网络节点，租户网络通信全部由计算节点分布式路由提供，网络节点关闭后，分布式高可用路由状态如下：

```

[root@controller1 ~]# neutron l3-agent-list-hosting-router dvr_ha_router
+-----+-----+-----+-----+-----+
| id | host | admin_state_up | alive | ha_state |
+-----+-----+-----+-----+-----+
| 7509137d-eda6-471c-84df-76fc15814666 | network1 | True | xxx | standby |
| c4258f65-1cec-4916-8d01-b84d3bea247e | compute2 | True | :-) | standby |
| 71a2efed-ee93-4721-9372-51bd099f3e5a | network2 | True | xxx | standby |
| b8e0fc29-e196-4b4a-8db7-842a4bbb2cf1 | compute1 | True | :-) | standby |
+-----+-----+-----+-----+-----+

```

访问 dvr_ha_server1 和 dvr_ha_server2 实例，通过 dvr_ha_server1 的 FixedIP (192.168.1.5) 访问 dvr_ha_server2 的 FixedIP (192.168.2.5)，同时通过 dvr_ha_server2 的 FixedIP (192.168.2.5) 反向访问 dvr_ha_server1 的 FixedIP (192.168.1.5)。正常情况下，两个实例彼此可以相互访问，测试结果如图 9-71 和图 9-72 所示。

```
# hostname
dvr-ha-server2
# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        inet6 ::1/128 scope host
            valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1454 qdisc pfifo_fast qlen 1000
    link/ether fa:16:3e:3e:1a:ca brd ff:ff:ff:ff:ff:ff
    inet 192.128.2.5/24 brd 192.128.2.255 scope global eth0
        inet6 fe80::f816:3eff:fe3e:1aca/64 scope link
            valid_lft forever preferred_lft forever
# ping 192.128.1.5
PING 192.128.1.5 (192.128.1.5): 56 data bytes
64 bytes from 192.128.1.5: seq=0 ttl=63 time=3.670 ms
64 bytes from 192.128.1.5: seq=1 ttl=63 time=0.822 ms
64 bytes from 192.128.1.5: seq=2 ttl=63 time=1.024 ms
64 bytes from 192.128.1.5: seq=3 ttl=63 time=2.274 ms
64 bytes from 192.128.1.5: seq=4 ttl=63 time=0.819 ms
```

图 9-71 dvr_ha_server2 正常访问 dvr_ha_server1

```
# hostname
dvr-ha-server1
# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        inet6 ::1/128 scope host
            valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1454 qdisc pfifo_fast qlen 1000
    link/ether fa:16:3e:55:04:51 brd ff:ff:ff:ff:ff:ff
    inet 192.128.1.5/24 brd 192.128.1.255 scope global eth0
        inet6 fe80::f816:3eff:fe55:451/64 scope link
            valid_lft forever preferred_lft forever
# ping 192.128.2.5
PING 192.128.2.5 (192.128.2.5): 56 data bytes
64 bytes from 192.128.2.5: seq=0 ttl=63 time=0.913 ms
64 bytes from 192.128.2.5: seq=1 ttl=63 time=1.453 ms
64 bytes from 192.128.2.5: seq=2 ttl=63 time=0.777 ms
64 bytes from 192.128.2.5: seq=3 ttl=63 time=2.499 ms
64 bytes from 192.128.2.5: seq=4 ttl=63 time=1.110 ms
```

图 9-72 dvr_ha_server1 正常访问 dvr_ha_server2

3) 分布式高可用路由 FloatingIP 实例 DNAT/SNAT 高可用验证。

对于具有 FloatingIP 的实例，其东西向和南北向的网络通信均无须网络节点参与，网络节点全部故障的情况下，实例东西向通信功能在步骤 2 中已经验证，这里主要验证 DVR/L3 HA 网络高可用模式下 FloatingIP 实例的南北向通信，包括外网访问实例 (DNAT) 和实例访问外网 (SNAT) 两种通信方式。为了验证分布式高可用路由 FloatingIP 实例中，网络 L3 Service 功能完全由计算节点 L3 Agent 提供，现将网络节点 network1 和 network2 全部关闭，并为 dvr_ha_server1 和 dvr_ha_server2 绑定 FloatingIP 地址。网络节点关闭后，分布式高可用路由状态如下：

```
[root@controller1 ~]# neutron l3-agent-list-hosting-router dvr_ha_router
```

id	host	admin_state_up	alive	ha_state
7509137d-eda6-471c-84df-76fc15814666	network1	True	xxx	standby
c4258f65-1cec-4916-8d01-b84d3bea247e	compute2	True	:-)	standby
71a2efed-ee93-4721-9372-51bd099f3e5a	network2	True	xxx	standby
b8e0fc29-e196-4b4a-8db7-842a4bbb2cf1	compute1	True	:-)	standby

为实例 dvr_ha_server1 和 dvr_ha_server2 绑定 FloatingIP 后, 实例状态如下:

```
[root@controller1 ~]# nova floating-ip-associate dvr_ha_server1 192.168.115.207
[root@controller1 ~]# nova floating-ip-associate dvr_ha_server2 192.168.115.208
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State	Networks
...ac5	dvr_ha_server1	ACTIVE	-	Running	admin-net1=192.128.1.5, 192.168.115.207
...944	dvr_ha_server2	ACTIVE	-	Running	admin-net2=192.128.2.5, 192.168.115.208

实例绑定 FloatingIP 后, 从外网 (192.168.115.0/24) 中的任意主机通过 SSH 以 Floating-IP 的形式访问 dvr_ha_server1 和 dvr_ha_server2, 正常情况下, 用户从外网 (如控制节点, 外网 IP 地址为 192.168.115.35) 可以直接通过实例 FloatingIP 访问实例, 测试结果如图 9-73 所示。此外, 正常情况下实例 dvr_ha_server1 和 dvr_ha_server2 可以正常访问外网 (如 Internet), 测试结果如图 9-74 所示。

```
[root@controller1 ~]# ssh 192.168.115.207
root@192.168.115.207's password:
# hostname
dvr-ha-server1
# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        inet6 ::1/128 scope host
            valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1454 qdisc pfifo_fast qlen 1000
    link/ether fa:16:3e:55:04:51 brd ff:ff:ff:ff:ff:ff
    inet 192.128.1.5/24 brd 192.128.1.255 scope global eth0
        inet6 fe80::f816:3eff:fe55:451/64 scope link
            valid_lft forever preferred_lft forever
# exit
connection to 192.168.115.207 closed.
[root@controller1 ~]# ssh 192.168.115.208
root@192.168.115.208's password:
# hostname
dvr-ha-server2
# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        inet6 ::1/128 scope host
            valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1454 qdisc pfifo_fast qlen 1000
    link/ether fa:16:3e:55:04:51 brd ff:ff:ff:ff:ff:ff
    inet 192.128.2.5/24 brd 192.128.2.255 scope global eth0
        inet6 fe80::f816:3eff:fe55:451/64 scope link
            valid_lft forever preferred_lft forever
```

图 9-73 控制节点 SSH 正常访问 dvr_ha_server1 和 dvr_ha_server2 实例


```

[root@controller1 ~]# ssh 192.168.115.208
root@192.168.115.208's password:
# hostname
dvr-ha-server2
# ping www.baidu.com
PING www.baidu.com (180.97.33.107): 56 data bytes
64 bytes from 180.97.33.107: seq=0 ttl=126 time=42.275 ms
64 bytes from 180.97.33.107: seq=1 ttl=126 time=42.421 ms
64 bytes from 180.97.33.107: seq=2 ttl=126 time=42.795 ms
64 bytes from 180.97.33.107: seq=3 ttl=126 time=42.858 ms
^C
-- www.baidu.com ping statistics --
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 42.275/42.587/42.858 ms
# exit
Connection to 192.168.115.208 closed.
[root@controller1 ~]# ssh 192.168.115.207
root@192.168.115.207's password:
# hostname
dvr-ha-server1
# ping www.baidu.com
PING www.baidu.com (180.97.33.107): 56 data bytes
64 bytes from 180.97.33.107: seq=0 ttl=126 time=75.702 ms
64 bytes from 180.97.33.107: seq=1 ttl=126 time=41.726 ms
64 bytes from 180.97.33.107: seq=2 ttl=126 time=42.209 ms
64 bytes from 180.97.33.107: seq=3 ttl=126 time=42.327 ms
^C
-- www.baidu.com ping statistics --
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 41.726/50.491/75.702 ms

```

图 9-74 FloatingIP 实例正常访问 Internet

从图 9-73 和图 9-74 所示中可以看到,在为实例绑定了 FloatingIP 后,通过 DVR 高可用路由,即使网络节点 network1 和 network2 全部故障,实例 dvr_ha_server1 和 dvr_ha_server2 与外网之间的南北网络通信仍然不受影响,同时由于实例东西网络通信也不经过网络节点,因此两个不同租户子网中的实例彼此之间的访问也不会受到影响。

上述的验证结果充分说明,在 Mitaka 版本中,DVR/L3 HA 部署方案是切实可行的网络高可用方案,此方案结合了 L3 HA 和 DVR 高可用方案的各自优点,既解决了传统 L3 HA 网络架构下网络节点所带来的性能瓶颈问题,又解决了 DVR 网络架构下网络节点 L3 SNAT 单点故障的问题。因此,可以认为 Mitaka 版本推出的 DVR/L3 HA 部署方案是真正接近生产环境的全方位网络高可用部署方案,通过 DVR 兼容 L3 HA 方案的实施,OpenStack 集群中 Neutron 网络的性能和扩展性问题,以及网络服务的高可用问题都可以通过 Neutron 的原生分布式高可用方案得到解决,从而避免了复杂的三方集群管理软件的使用。

9.9 本章小结

网络服务是云计算三大核心项目之一,而作为网络服务项目 Neutron 是 OpenStack 中最为核心的项目,由于网络虚拟化本身的复杂性,Neutron 项目的成熟度依然不如计算服务 Nova。因此,Neutron 项目发展极为迅速,新功能不断完善,并且越来越多的用户正将 Neutron 虚拟化网络用于生产环境中。本章对 Neutron 的架构和内部组件进行了深入分析,并对目前使用最为广泛的 Neutron 网络类型及其实现过程进行了详细阐述。此外,作为生产环境的核心虚拟化网络,本章对 Neutron 网络的各种高可用方案进行了部署实现和验证,并对各种高可用方案下实例数据的访问过程进行了详细讲解。通过本章的阅读,读者不仅可以了解 Neutron 项目的原理架构,还可以根据本章的讲解在生产环境中部署高可用的 OpenStack 网络。

OpenStack 存储服务

存储、计算和网络是任何 IaaS 基础架构都必须实现的核心部分，OpenStack 作为目前最为成功的开源云计算项目，在成立之初便通过 Swift 和 Nova-Volume 提供了对象存储和块存储服务。到 Mitaka 版本发布为止，OpenStack 存储项目已经发展成为以 Swift、Cinder（由 Nova-Volume 独立而来）和 Manila 为主，覆盖对象存储、块存储和文件系统存储的服务。而在这些存储服务中，以 Swift 和 Cinder 最为成熟，并且以 Cinder 块存储服务使用最为广泛。由于 Cinder 的开放式插件设计架构，使得各种基于商业硬件存储阵列和开源软件实现的存储方案都可以成为 Cinder 的存储后端，因此围绕 Cinder 的存储生态圈非常活跃，几乎全部的存储厂商都在向社区提交 Cinder 插件，而用户完全可以选用适合自己的存储插件而不用担心厂商锁定的困扰。Cinder 项目很好地实现了活跃的上游存储厂商与终端用户自主选择的结合。在 Cinder 的众多后端开源存储实现方案中，Ceph 作为一种集合了对象存储、块存储和文件系统存储的统一分布式存储集群，由于其独特的数据寻址和动态分布式存储方式，其作为独立的存储开源项目在 OpenStack 社区有着极为广泛的使用。Ceph 除了可以作为 Cinder 的后端提供块存储服务外，还可以直接集成到 Nova 和 Glance 以提供 RBD 存储服务，同时 Ceph 也可以作为 Swift 后端通过 Ceph RGW 提供对象存储服务，因此，Ceph 实现了与 OpenStack 存储服务的无缝对接，并以其高扩展性、高可用性和高性能获得了 OpenStack 社区的极大支持。本章将对 OpenStack 的 Cinder 块存储服务和开源存储集群 Ceph 进行架构原理介绍，并以实战的形式对二者进行实施讲解，同时将 Ceph 与 OpenStack 进行集成实现。通过对本章的阅读，读者对 OpenStack 的存储架构将会有比较全面的了解，并能够成功地将 Ceph 集成到 OpenStack 中以提供各种类型的数据存储服务。

10.1 OpenStack 存储概述

10.1.1 OpenStack 存储分类对比

任何 IT 系统为了正常运行并存储数据，都必须能够访问存储设备。在 OpenStack 中，存储可以分为临时存储（Ephemeral Storage）和持久性存储（Persistent Storage）。临时存储设备并不保存其上存储数据的改变，当用户释放临时存储设备时，其又恢复到最初的原始状态（位于其上的数据将会丢失），而持久性存储设备将会永久性保存其上数据的改变。如果用户仅部署 OpenStack 计算服务（Nova），则默认情况下用户并不能访问持久性存储设备，Nova 创建的虚拟机实例所关联的磁盘是“临时性”的（在 OpenStack 中由 Flavor 设定，即通常所谓的虚拟机系统盘），从用户角度而言，当虚拟机被用户终止后，“临时性”磁盘上的数据也将丢失。与临时性存储不同，持久性存储设备的使用寿命独立于任何其他系统设备或资源，并且其上存储的数据一直可用，因此，当 OpenStack 实例终止后，持久性存储设备上的数据也仍然可用。从用户生产系统角度而言，OpenStack 中使用最多的是持久性存储设备，这也是本章重点介绍和讨论的部分。到目前为止，OpenStack 支持三种类型的持久性存储，即块存储、对象存储和文件系统存储。其中，块存储是使用最为普遍的持久性存储，同时也是支持的开源存储技术和厂商最多的存储类型，因此本章将重点介绍 OpenStack 中的块存储 Cinder 项目和分布式存储集群 Ceph 中的块存储部分在 OpenStack 高可用集群部署中的应用。

1. 块存储

块存储（Block Storage）是生产环境中使用最多的数据存储类型，在传统 IT 架构中，关键业务生产系统几乎都是通过访问块存储形式来存储数据的，而存储厂商的存储阵列通常经过 SAN 存储网络以卷（Volume）或逻辑存储单元（Logic Unit, LUN）的形式将块存储挂载到操作系统上，用户在操作系统上可以对块存储设备进行分区（Partition）、格式化（Formation）和挂载（Mount）等操作，而应用程序以文件系统或者裸设备的形式访问块存储设备。在 OpenStack 云计算环境中，块存储的实现和访问形式与传统 IT 架构类似，只不过云环境中的块存储（通常称为云盘）被挂载到虚拟机实例中供用户使用，即用户通过将 Volumes 挂载（Attach）到虚拟机上以实现块存储设备的交互。

在 OpenStack 中，块存储设备具有持久性。用户可以将 Volumes 从实例上卸载（Detach）并重新挂载（Attach）到其他实例上继续使用，而 Volumes 中的数据对于不同实例主机均可访问，并且当实例故障或被用户终止后，用户存储在块存储 Volumes 上的数据并不会丢失且可以通过其他实例继续访问。块存储在 OpenStack 中通过 Cinder 项目来实现，并且 Cinder 支持不同形式的多种后端存储驱动，用户只需选择 Cinder 块存储驱动所支持的存储后端即可。通常情况下，为了提高整体的读写 IO，实例应用程序通过块存储设备驱动直接访问底层硬件存储块设备，不过 Cinder 也支持使用文件系统来模拟创建 Volumes 并挂载到实例供应用程序访问，如 NFS 和 GlusterFS 存储后端便是基于文件系统

的块存储实现的。

2. 对象存储

使用对象存储，用户可以通过 REST API 来访问二进制存储对象，而无须如块存储一样将存储设备挂载到实例才能与其进行交互，亚马逊 AWS 的 S3 存储便是最为成熟的商业对象存储。OpenStack 中实现对象存储的项目是 Swift，这也是 OpenStack 最早的项目之一，如果用户需要存储或管理大量无序、非结构化的数据集，则可以考虑部署对象存储 Swift。此外，作为存储镜像文件的替换方案，用户也可以将 OpenStack 的镜像存储到对象存储中。OpenStack 对象存储消除了传统文件系统的部分约束，并提供了具有极高的扩展性和高可用性的存储解决方案^①。通常，对象存储集群通过将数据以多份拷贝存储的形式扩散到不同的存储节点以实现存储集群的可持续性和可用性，而非借助硬件层面的高可用性来实现存储数据访问的持续性和可用性，因此对象存储集群中的每一份数据通常具有多份拷贝（默认情况一般为 3 份），用户可以更改数据拷贝的份数以提高数据存储的高可用性，但是过多的存储重复数据必然导致物理存储资源利用率较低。此外，影响到对象存储集群数据高可用的另一个因素便是数据库的放置位置和方式，如将同一份数据的不同拷贝放置到位于不同故障域的存储节点上，则可以提高存储集群在故障发生时的数据可用性。

3. 文件系统存储

共享文件系统服务为多租户云环境提供了一系列管理共享文件系统的服务集，用户通过挂载远程文件系统到本地实例，以进行文件存储和交互的形式实现与共享文件系统服务的交互。共享文件系统服务为用户提供了“共享”，这里的“共享”通常指一个远程可挂载的文件系统，在同一时刻不同用户的多个主机实例可以同时挂载并访问同一个共享。通过 OpenStack 中的共享文件系统服务，用户可以创建一个共享，指定共享的大小、访问协议和可见级别。根据选取的后端模式和有无共享网络，用户可以在共享或独立服务器上创建共享，同时还可以将不同的共享集合到共享组中以保持多个共享数据的一致性。此外，用户还可以为共享或共享组创建快照，也可以由快照创建共享。创建共享之后，用户可以设置共享的限额并查看共享资源的使用情况。与块存储类似，共享文件系统服务也具有持久性，共享文件系统可以挂载到任意多的客户机上，共享文件系统可以从某个实例卸载并重新挂载到其他实例，而在这个过程中共享文件系统上的数据不会改变。在 OpenStack 中，实现共享文件系统服务的项目是 Manila，Manila 以驱动的形式支持多种后端，用户可以通过配置共享文件系统服务以实现从一个或多个后端提供共享，而共享服务器通常是通过各种不同协议，如 NFS、CIFS、GlusterFS 或 HDFS 导出共享文件的虚拟机。

OpenStack 中不同的存储服务提供了不同的存储设备，用户可以根据自己的需求部署不同的存储服务。表 10-1 列举了 OpenStack 不同存储服务之间的差异，用户可以将自己的

① 关于 Swift 对象存储更详细的资料请参考：http://docs.openstack.org/developer/swift/overview_architecture.html。

存储需求与不同 OpenStack 存储服务的特性集合起来以选择最适合自己的 OpenStack 存储服务。

表 10-1 OpenStack 中的存储概念

	主要用途	访问方式	访问客户端	管理服务	数据周期	决定存储设备大小因素	典型使用案例
临时存储	运行操作系统和提供启动空间	通过文件系统访问	虚拟机	Nova	虚拟机终止	Flavors 指定	虚机中第一块盘 10GB，第二块盘 20GB（类似 Windows 系统中的 C 盘和 D 盘）
块存储	为虚拟机挂载额外的永久性存储	块设备可以被分区、格式化和挂载访问	虚拟机	Cinder	用户删除	用户创建时指定	1TB 磁盘（外置存储，如 Windows 系统中的移动硬盘）
对象存储	存储海量数据集，包括虚拟机镜像	REST API	任何客户端	Swift	用户删除	可用物理存储空间和数据副本数目	AWS S3
共享文件系统存储	为虚拟机挂载额外的永久性存储	分区、格式化和挂载访问	虚拟机	Manila	用户删除	用户创建时指定或设置的限额	NFS

10.1.2 OpenStack 存储后端选择

不同的云资源使用者通常会根据其自身的业务特性提出不同的需求，而 OpenStack 提供了不同的存储后端供用户选择。通常，在选择 OpenStack 存储后端时，用户应该考虑以下几个问题：

- ☐ 是否需要使用块存储？
- ☐ 是否需要使用对象存储？
- ☐ 是否需要支持 live-migration 迁移？
- ☐ 是否利用计算节点上的持久性存储驱动提供存储资源还是独立的外部存储？
- ☐ 哪种存储方案可以实现最佳的性能与成本控制？
- ☐ 哪种存储方案更易于操作管理？
- ☐ 怎样实现存储的冗余和分布式？
- ☐ 如果存储节点故障可能发生什么情况？
- ☐ 灾难发生时数据丢失的可能性有多大？

作为开源云计算项目，用户可以选用不同的开源存储后端软件结合纯粹的商业裸机服务器来实现 OpenStack 后端存储。除了 OpenStack 集成开发的 Cinder 和 Swift 项目之外，在开源存储领域仍然有很多非常流行和成熟的开源存储方案，如 Ceph、GlusterFS、Sheepdog 等等。表 10-2 所示是各种主流开源存储解决方案对不同存储类型的支持情况。

除了开源存储解决方案之外，用户还可以选择商业存储解决方案作为 OpenStack 集群的存储后端。由于 Cinder 项目以存储 Driver 插件的形式来管理不同的存储后端实现（Cinder 与 Neutron 类似，都被设计为可插拔架构），很多存储厂商都实现了统一的 Cinder 存储 Driver 接口，如 IBM、EMC、Netapp、HPE、Hitachi 和华为等很多存储厂商都不同程度地实现了对 Cinder 后端存储的支持。表 10-3 所示为 OpenStack 对不同存储后端的版本支持矩阵。注意，支持矩阵是向前兼容的，如 Kilo 版本支持的存储后端在 Mitaka 和 Newton 版本中通常也被支持。

[illegible]

(续)

[illegible]

(续)

序号	驱动名称	创建卷	删除卷	卷挂载	卸载卷	卷扩展	创建快照	删除快照
50	Violin Memory	Kilo	Kilo	Kilo	Kilo	Kilo	Kilo	Kilo
51	VMware	Havana	Havana	Havana	Havana	Icehouse	Havana	Havana
52	Windows Server 2012	Grizzly	Grizzly	Grizzly	Grizzly		Grizzly	Grizzly
53	X-IO technologies	Kilo	Kilo	Kilo	Kilo	Kilo	Kilo	Kilo
54	Zadara Storage	Folsom	Folsom	Folsom	Folsom	Havana	Havana	Havana
55	Solaris (ZFS)	Diablo	Diablo	Diablo	Diablo			
56	ProphetStor (Flexvisor)	Juno	Juno	Juno	Juno	Juno	Juno	Juno
57	Infotrend	Liberty	Liberty	Liberty	Liberty	Liberty	Liberty	Liberty
58	CoprHD	Newton	Newton	Newton	Newton	Newton	Newton	Newton
59	Kaminario	Newton	Newton	Newton	Newton	Newton	Newton	Newton

对于 OpenStack 用户而言，是选择如 LVM 这类开源的存储驱动后端，还是选择存储设备厂商（Vendors）驱动后端，需要根据用户自己对云环境中存储资源的使用情况来决定。通常而言，商业存储解决方案具有更多的数据操作功能，如数据去重（De-Duplication）、数据压缩（Compression）、数据瘦身（Thin-Provisioning）和数据分级（Tier）。此外，商业存储对 RAID 技术的支持更为成熟并且不像服务器 RAID 卡一样会受到很多限制，同时商业存储的远程数据复制和镜像等功能为数据冗余和容灾提供了更多的选择。但是，不可否认的是商业存储采购费用和后期维护成本都极为可观，并且不可避免地会受厂商锁定（Vendor Locked-In）的困扰，尤其是在开源技术不断普及和自主可控呼声不断上涨的时下，采用商业存储方案无疑会使自己的技术团队陷入按部就班的“步骤式”成长之中，而这种情况非常不利于企业的自主创新。相反，如果采用开源存储解决方案，则用户前期需要更多的人力成本投入，而且很多开源存储方案并不如商业解决方案成熟，为了适应自身环境，用户必须在完全理解的基础之上对相应的开源软件进行定制化，加之开源软件版本变更过于频繁，对用户而言，任何的版本更新和定制化都是一次存在风险的尝试。因此，采用开源存储解决方案对于企业技术团队而言将会是一个挑战。然而挑战之后必然是对技术的自主可控和灵活应变，所以开源存储解决方案更多的适合成本预算有限、技术实力充分且不想依赖厂商或被厂商锁定的企业。

从目前 OpenStack 社区发展情况来看，采用开源 Ceph 存储集群作为 Cinder 存储后端似乎赢得了更多客户的青睐。Ceph 作为一种统一了对象存储、块存储和文件存储的分布式存储集群，其不仅可以持久性存储 OpenStack 镜像和块存储数据，还可以作为 Nova 创建虚拟机时的临时存储，因此 Ceph 满足了 OpenStack 对于不同存储类型的需求。本章后续内容将会重点介绍 OpenStack 的 Cinder 块存储项目和分布式存储集群 Ceph，以及如何将 Ceph 集成到 OpenStack 中。

10.2 Cinder 块存储

10.2.1 Cinder 块存储架构

OpenStack 集群中的存储通常分为块存储、对象存储和文件系统存储，简单而言，块存储就是通过 SAN 或 iSCSI 等存储协议将存储设备端的卷（Volume）挂载到虚拟机上并进行分区和格式化，然后挂载到本地文件系统使用的存储实现方式；而文件系统存储则是通过 NFS 或 CIFS（Samba）等网络文件系统协议将远程文件系统挂载到本地系统使用的存储实现方式；相对而言，对象存储在实现和使用方式上与块存储和文件系统存储都不同，对象存储是一种以 REST API 方式提供数据访问的存储实现方式。在 OpenStack 中，块存储是使用最多的数据存储实现方式，并且由 Cinder 项目提供，Cinder 是 OpenStack 集群中提供块存储服务的独立项目，其前身为 Nova 项目中的 Nova-Volume 子项目，并在 OpenStack 的 F 版本后独立成为 OpenStack 的核心项目。

在 OpenStack 中，块存储服务 Cinder 为 Nova 项目所实现的虚拟机实例提供了数据持久性的存储服务。此外，块存储还提供了 Volumes 管理的基础架构，同时还负责 Volumes 的快照和类型管理。从功能层面来看，Cinder 以插件架构的形式为各种存储后端提供了统一 API 访问接口的抽象层实现，使得存储客户端可以通过统一的 API 访问不同的存储资源，而不用担心底层各式各样的存储驱动。Cinder 提供的块存储通常以存储卷的形式挂载到虚拟机后才能使用，目前一个 Volume 同时只能挂载到一个虚拟机，但是不同的时刻可以挂载到不同的虚拟机，因此 Cinder 块存储与 AWS 的 EBS 不同，不能像 EBS 一样提供共享存储解决方案。除了挂载到虚拟机作为块存储使用外，用户还可以将系统镜像写入块存储并从加载有镜像的 Volume 启动系统（SAN BOOT）。Cinder 块存储服务主要由以下几部分组成。

- ❑ Cinder-api：Cinder-api 是一种 WSIG 类型的应用服务，其主要负责接收来自 Horizon 或命令行客户端的块存储 API 请求，同时负责请求客户端的身份信息验证（通过 Keystone 项目实现）。Cinder-api 接收到客户端请求后，根据 Cinder-scheduler 的存储后端调度结果，将请求 API 路由到运行 Cinder-volume 服务的对应后端存储上。
- ❑ Cinder-scheduler：与 Nova-scheduler 的功能类似，Cinder-scheduler 是 Cinder 项目的后端 Volumes 服务调度器，当 Cinder-api 接收到客户端请求后，将由 Cinder-scheduler 服务来负责 API 的路由。根据用户配置的 Scheduler 策略，Cinder-api 请求可以采用形如 Round-robin 的轮询方式路由到运行 Volume 服务的各个存储节点，也可以采用 FilterScheduler 来实现更为复杂和智能的后端存储节点过滤策略。在 Cinder 的配置中，FilterScheduler 是默认设置，通过 FilterScheduler 的配置可以实现基于 Capacity、Availability Zone、Volume Types、Capabilities 或者用户自定义过滤策略的后端存储节点调度。
- ❑ Cinder-volume：Cinder-volume 是 Cinder 项目中真正提供块存储和不同存储驱动插件管理的服务，不论节点处于什么角色，只要其运行 Cinder-volume 服务，该节点

均可称为存储节点。Cinder-volume 服务通过 AMQP 与 Cinder-scheduler 进行交互，将其所管理的各个存储驱动后端的运行、性能和容量等参数实时传递到 Cinder-scheduler，以便 Cinder-scheduler 根据这些参数进行存储节点的调度。此外，Cinder-volume 通过 Driver 架构实现与不同存储驱动后端的交互。

- ❑ Cinder-backup：Cinder-backup 为块存储卷（Volumes）提供了备份服务，要实现 Volumes 的备份，需要提供备份存储驱动的实现，目前比较常见的备份存储后端由 Swift 提供。与 Cinder-volume 类似，Cinder-backup 也通过 Driver 插件架构的形式与不同的存储备份后端交互。
- ❑ 消息队列：Cinder 项目的不同内部服务组件之间通过 Queue 进行消息交互，如 Cinder-volume 与 Cinder-scheduler 之间的信息交互。Cinder 项目内部各个服务之间的消息交互如图 10-1 所示。

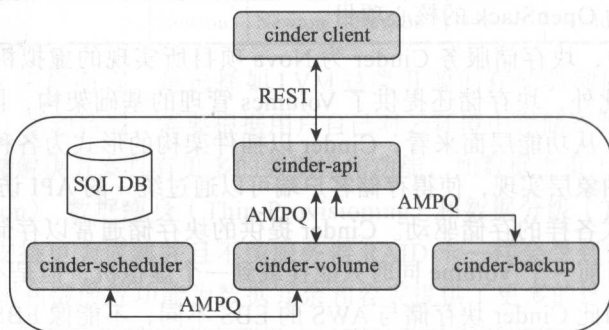


图 10-1 Cinder 内部服务交互

Cinder 项目的整体架构如图 10-2 所示，图形界面客户端或命令行客户端通过 Horizon 与 Cinder-api 服务交互，Cinder-api 通过 Keystone 实现客户端身份验证，Cinder-volume 通过 Driver 插件形式管理各种 Volume Providers 存储后端，并将 Volumes 相关的消息写入 DB 进行保存。

10.2.2 Cinder 块存储使用

Cinder 提供了不同存储后端的抽象统一接口，通过 Cinder 项目，用户可以将不同的存储后端整合在一起并通过统一的 OpenStack API 接口对外提供存储资源服务。而 Cinder 块存储最常见的使用方式便是在不同的存储后端上创建 Volumes，并将 Volumes 挂载到虚拟机上以块存储的形式提供存储服务。此外，为了实现虚拟机的高可用，通常也将系统镜像写入 Cinder 块存储，使其成为 Bootable Volume，从而可以在虚拟机宿主服务器故障的情况下在其他物理主机上迅速重启位于故障物理机上的虚拟机。图 10-3 所示为 Nova 虚拟机与 Cinder Volume 的挂载使用示例，用户根据需求在 Cinder 的存储池中创建所需大小的 Volume，然后将 Volume 挂载到指定的虚拟机上。一个 Volume 同一时刻只能挂载到某

个虚拟机，但是不同的多个 Volume 可以同时挂载到某个虚拟机，并且在 Volume 被卸载掉 (Detach) 之后其可以任意挂载到其他虚拟机。当用户将系统镜像写入 Volume 之后，此 Volume 成为 Bootable Volume，用户可利用 Bootable Volume 创建虚拟机。

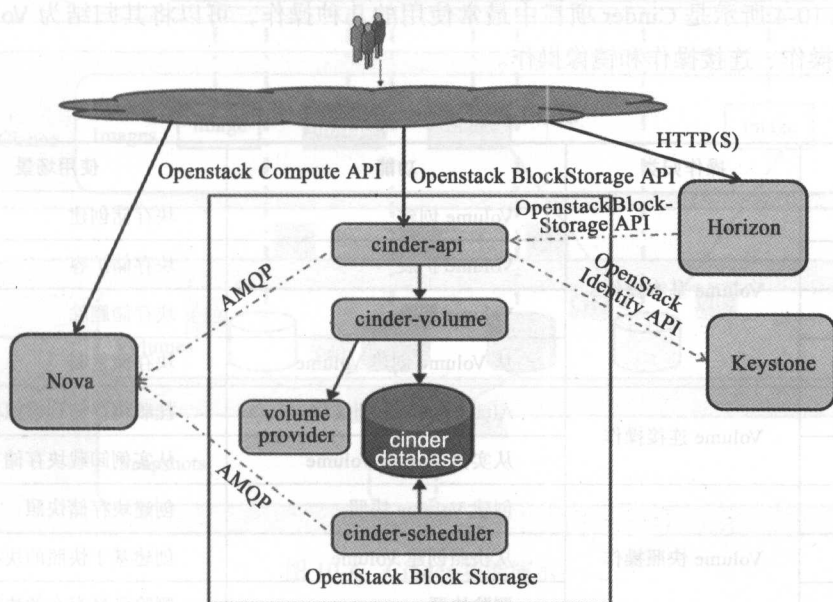


图 10-2 Cinder 项目架构

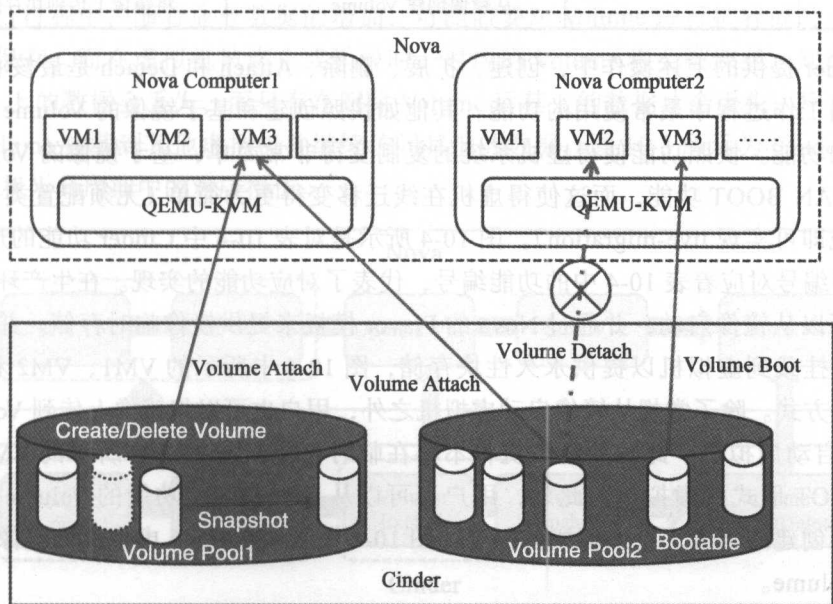


图 10-3 Volume 在 Nova 虚拟机中的挂载使用

除了将 Volume 关联到虚拟机 (Attach to Instance) 和从虚拟机卸载 Volume (Detach from Instance) 之外, Cinder 还提供了很多块存储操作功能, 如基于 Snapshot 的 Volume 创建、基于 Image 的 Volume 创建、基于 Volume 的 Volume 创建、对 Volume 进行 Extend 和 Backup 等操作。表 10-4 所示是 Cinder 项目中最常使用的几种操作, 可以将其归结为 Volume 基本操作、快照操作、连接操作和镜像操作。

表 10-4 Cinder 块存储操作

功能编号	操作归类	功能	使用场景
1	Volume 基本操作	Volume 创建	块存储创建
2		Volume 扩展	块存储扩容
3		Volume 删除	块存储删除
4		从 Volume 创建 Volume	块存储复制
5	Volume 连接操作	Attach Volume 到实例	挂载块存储到虚拟机
6		从实例 Detach Volume	从实例卸载块存储
7	Volume 快照操作	创建 Volume 快照	创建块存储快照
8		从快照创建 Volume	创建基于快照的块存储
9		删除快照	删除已经存在的快照
10	Volume 镜像操作	从 Volume 创建镜像	提取块存储中的镜像
11		从镜像创建 Volume	将镜像上传到块存储中

在 Cinder 提供的上述操作中, 创建、扩展、删除、Attach 和 Detach 是最核心的功能, 也是块存储工作过程中最常使用的功能。其他如快照创建和基于镜像的 Volume 创建也是比较重要的功能。快照功能使得虚机系统的复制变得非常简单, 基于镜像的 Volume 创建可以实现 SAN BOOT 功能, 而这使得虚机在线迁移变得更为简单 (无须配置类似 NFS 共享文件系统即可实现 live-migration)。图 10-4 所示是对表 10-4 中 Cinder 功能的形象描述, 图中的数字编号对应着表 10-4 中的功能编号, 代表了对应功能的实现。在生产环境中, 虚拟机系统可以从镜像启动, 并通过 Nova 的 Flavor 模板来提供镜像临时存储, 并将 Cinder 的 Volumes 挂载到虚拟机以提供永久性块存储, 图 10-4 中所示的 VM1、VM2 和 VM3 便是这种使用方式。除了常规从镜像启动虚拟机之外, 用户也可以将镜像上传到 Volume, 并从 Volume 启动虚拟机, 此时的虚拟机将不存在临时存储, 图 10-4 中所示的 VM4 便是这种 SAN BOOT 形式的虚拟机。此外, 用户还可以从具有 BOOT 功能的 Volume 中创建镜像, 并利用创建的镜像进行虚拟机的创建, 图 10-4 中所示的 VM5 虚拟机的镜像便是来自 Bootable Volume。

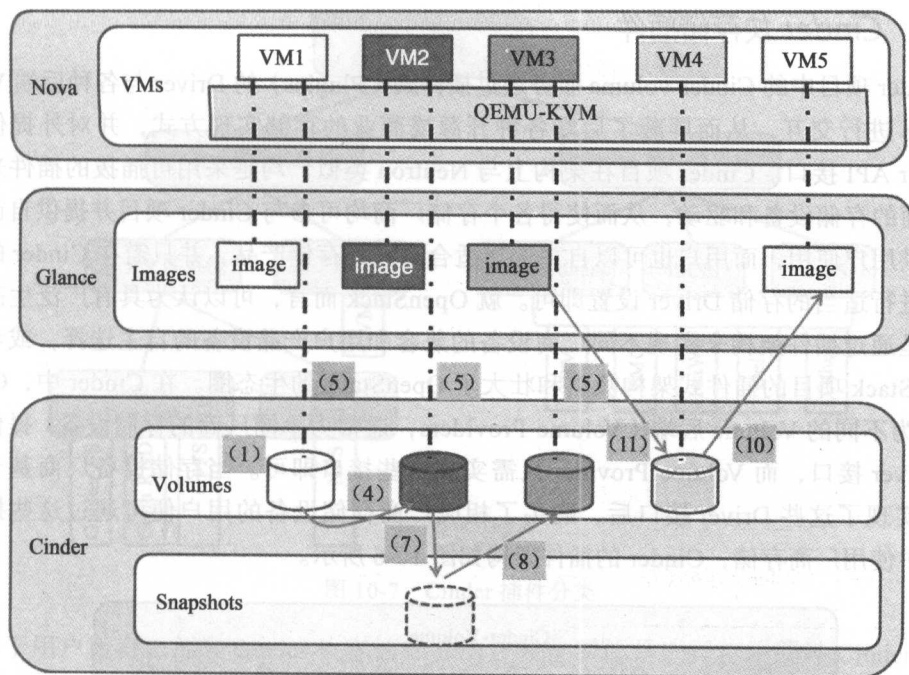


图 10-4 Cinder 功能演示

通常而言，Cinder 块存储提供的 Volume 总是与 Nova 虚拟机的生命周期密切关联在一起，如在虚拟机的创建时，对应着也要创建 Volume 并将 Volume 挂载到虚拟机。在虚拟机的运行过程中，随着业务数据的增加，可能需要对 Volume 进行扩容或者对主要数据进行快照保存，而在虚拟机被终止或销毁时，对应的 Volume 将会被卸载。卸载并不意味着 Volume 上的数据会丢失，而只有在删除 Volume 后其上的数据才会丢失。用户可以将卸载后的 Volume 挂载到其他虚拟机并继续访问其上的数据。图 10-5 所示为 Cinder 块存储在 Nova 虚拟机生命周期中的对应操作。

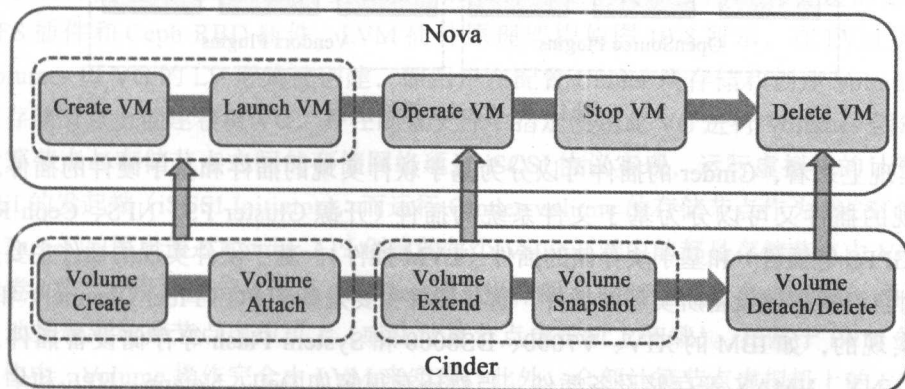


图 10-5 Nova 虚拟机与 Cinder 块存储生命周期对应关系

10.2.3 Cinder 块存储插件

Cinder 项目中的 Cinder-volume 服务通过插件式 (Plugins) 的 Driver 与各种后端 Volume Providers 进行交互, 从而屏蔽了后端各种开源或商业的存储实现方式, 并对外提供统一的 Cinder API 接口。Cinder 项目在架构上与 Neutron 类似, 均是采用可插拔的插件来兼容各个厂商的存储设备和驱动, 从而使得各个存储厂商均可参与 Cinder 项目并提供自己的存储插件供用户使用, 而用户也可以自主选择适合自己的存储产品, 并只需在 Cinder 的配置文件中进行适当的存储 Driver 设置即可。就 OpenStack 而言, 可以认为具有广泛生态圈的项目均是通过插件架构来实现不同厂商设备的兼容和用户产品设备的自主选择, 或者说正是 OpenStack 项目的插件式架构吸引和壮大了 OpenStack 的生态圈。在 Cinder 中, Cinder-volume 为不同的 Volume 后端 (Volume Providers, 通常为不同厂商的存储设备) 提供了统一的 Driver 接口, 而 Volume Provider 只需实现这些接口即可。当存储设备厂商基于自己的设备实现了这些 Driver 接口后, 购买了相应厂商存储设备的用户便可通过这些插件在 Cinder 中使用厂商存储, Cinder 的插件架构如图 10-6 所示。

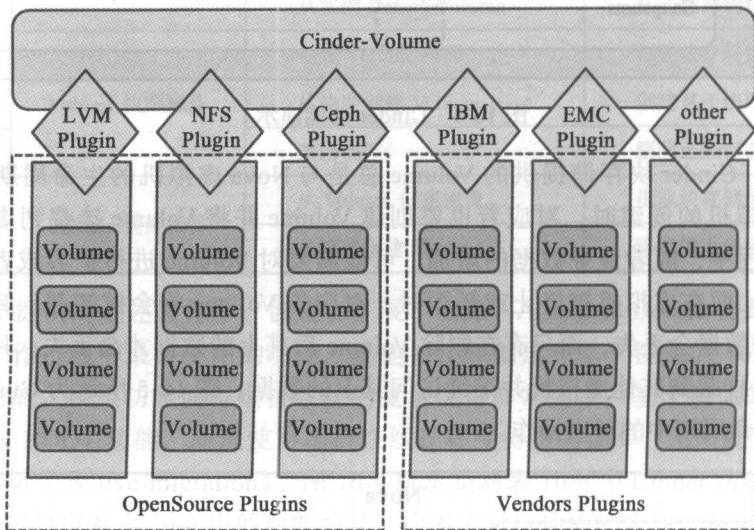


图 10-6 Cinder 插件架构

从实现上来看, Cinder 的插件可以分为基于软件实现的插件和基于硬件的插件。基于软件实现的插件又可以分为基于文件系统的插件 (开源 Gluster FS、NFS、Ceph RBD 和 IBM 的 GPFS 等插件) 和基于块存储的插件 (LVM 插件)。基于硬件实现的插件主要是各个厂商针对自己的存储设备所实现的插件, 这些插件主要是基于 FC (Fiber Channel) 和 iSCSI 协议来实现的, 如 IBM 的 XIV、V7000、DS8000 和 System Flash 等存储设备插件, 以及 EMC 的 VNX、VMAX 等存储设备插件, 当然还有很多如 Dell、Netapp、HPE 和华为等存储厂商所提供的插件。Cinder 插件分类如图 10-7 所示。

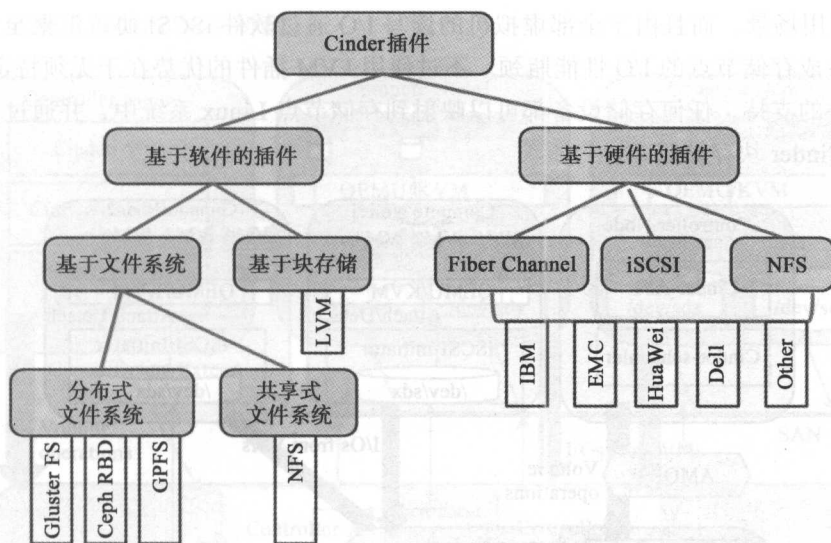


图 10-7 Cinder 插件分类

对于用户而言，部署 Cinder 块存储服务的首要步骤便是选择使用哪种 Cinder 存储后端插件，是采用完全基于开源软件实现的插件还是采用存储设备厂商提供的特定插件，需要根据用户自己的预算和特定的存储环境进行选择。通常 LVM 插件适用于各种厂商存储设备，但是其 I/O 性能相对较差，因此比较适合轻负载应用系统。而针对特定厂商存储设备的 FC 插件通常具有更好的 I/O 性能和较低的 I/O 延迟，但是需要采购特定厂商的存储设备，因此比较适合 I/O 要求较高的生产环境系统。除了基于本地 LVM 的 Cinder 插件和 Vendors 厂商插件外，日立（Hitachi）的 Mitsuhiro 团队还提出了一种集合 LVM 和 FC 的共享 LVM 插件，下文将主要对这几种插件进行介绍。

1. Local LVM 插件

基于软件实现的 Cinder 插件主要以开源软件为主，其中使用最为普遍的是 LVM 插件、NFS 插件和 Ceph RBD 插件。LVM 插件原理架构如图 10-8 所示，在 LVM 插件架构中，Volumes 以 VG 的 LV 形式被创建，即用户在配置 Cinder 块存储和创建 Volumes 之前，需要在存储节点上创建卷组 VG，并在配置文件中指定使用此 VG 进行 Volumes 创建源。此外，计算节点与存储节点之间的存储网络通过 iSCSI 协议实现，运行虚拟机的计算节点作为 iSCSI 的发起端（iSCSI Initiator），而运行 Cinder-volume 的存储节点作为 iSCSI 的目标端（iSCSI Target），用户通过 Cinder-API 创建的每个 Volume 对应的都是存储节点中 VG 上的一个逻辑卷 LV，当用户发起 Attach（或 Detach）操作时，存储节点上的 Volumes 通过 iSCSI 协议自动挂载到计算节点虚拟机上（或从计算节点虚拟机上卸载）。在基于 LVM 的 Cinder 创建架构中，Volume 操作完全由 LVM 来实现，此外，全部计算节点虚拟机上的 I/O 通过软件 iSCSI 协议集中传输到存储节点，因此采用 LVM 插件实现的 Cinder 块存储比较适合轻

负载 I/O 应用场景，而且由于全部虚拟机的读写 I/O 通过软件 iSCSI 协议汇聚至存储节点，这很容易造成存储节点的 I/O 性能瓶颈。不过使用 LVM 插件的优势在于无须特定存储设备和对应插件的支持，任何存储设备都可以映射到存储节点 Linux 系统中，并通过 LVM 插件架构实现 Cinder 块存储服务。

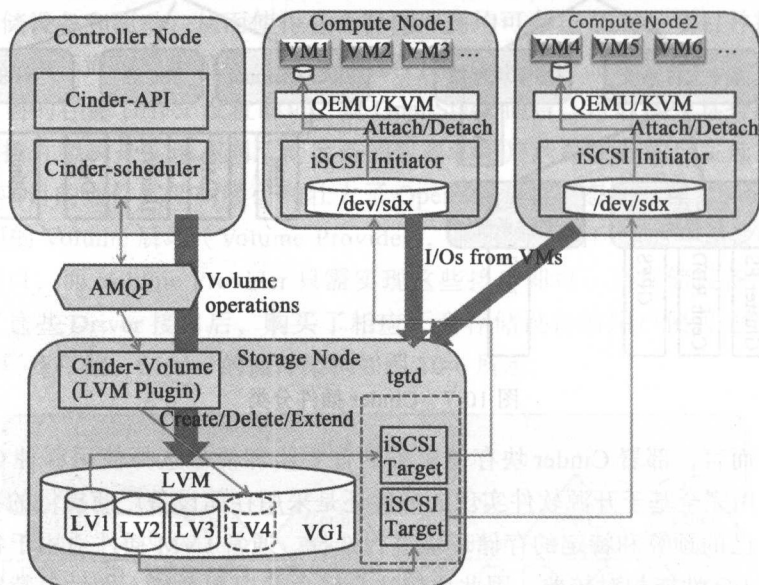


图 10-8 基于 Local LVM 插件的 Cinder 架构

2. Vendor FC 插件

由于 LVM 插件部署和使用简单，而且适用于不同的物理或虚拟存储设备，因此 LVM 插件是 OpenStack 官方 Cinder 部署文档指定的默认插件，但是基于 LVM 插件的 Cinder 块存储服务功能完全基于软件仿真来实现，因此仅适用于开发和测试等轻量级系统环境中。而在高并发 I/O 的生产环境中，基于 LVM 的 Cinder 服务并不具备高可用性和稳定性，尤其是 Cinder-Volume 服务在高可用实现方面一直是个难点。此外，由于虚拟机的 I/O 数据全部通过以太网写入存储节点，因此存储节点的性能和带宽限制很容易造成存储 I/O 瓶颈。对于要求实现高性能 I/O 和高稳定性的生产环境而言，采用特定存储 Vendors 所提供的 FC 插件是比较理想的选择。通常，各个存储厂商的中高端存储设备提供了更多的存储数据操作功能，如 Asynchronous/Copy-On-Write 数据快照、硬盘瘦身 (Thin-Provisioning)、存储数据自动分层 (Automated Storage Tier) 等高级功能。此外，采用厂商 FC 插件后，OpenStack 块存储服务的高可用性很大程度上转移到具备高度冗余的存储设备上，同时高端存储设备的控制器通常具有很高的前后端 I/O 带宽和数据处理能力，因此对于生产系统，通常建议购买支持 OpenStack 生态圈的特定存储设备厂商所生产的中高端存储设备，并采用该厂商所提供的 FC 插件来实现 Cinder 块存储服务。基于特定 Vendors 存储设备 FC 插件

的 Cinder 架构原理如图 10-9 所示。

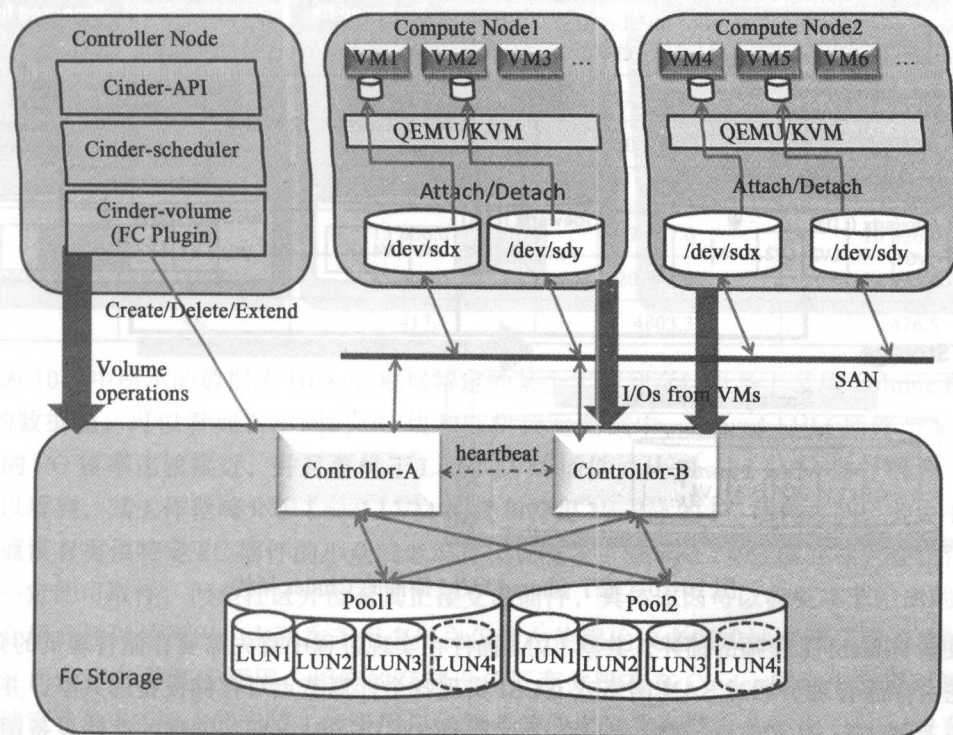


图 10-9 基于 Vendor FC 插件的 Cinder 架构

3. Shared LVM 插件

Local LVM 插件的优势在于使用简单，并且屏蔽了各种存储设备的差异。Volume 操作由 LVM 来实现，Volume 到计算节点虚拟机的挂载通过软件 iSCSI 实现。而其不足之处在于虚拟机 I/O 不能像 FC 插件一样直接写入存储设备，因此一旦 Cinder-volume 或 iSCSI Target 端服务故障，则虚拟机到存储节点的 I/O 将会停止，虚拟机也有可能崩溃，并且整个集群存储网络的带宽会受到运行 Cinder-volume 服务存储节点的带宽限制。针对 Local LVM 的这些不足，Hitachi 提出了增强型 LVM 插件，即 Shared LVM 插件^①。根据 Shared LVM 插件的设计思路和实现方式，Shared LVM 插件将不再需要 SCSI Target，虚拟机 I/O 通过 FC 网络直接访问存储设备，控制节点和不同的计算节点可以同时访问共享存储上的 VGs。用户通过 Cinder-API 可以对共享 LVM 进行 Volume 的创建、删除和挂载等操作。这种基于共享 LVM 的插件架构如图 10-10 所示。

① <https://wiki.openstack.org/wiki/Cinder/NewLVMbasedDriverForSharedStorageInCinder>
<https://blueprints.launchpad.net/cinder/+spec/lvm-driver-for-shared-storage>。

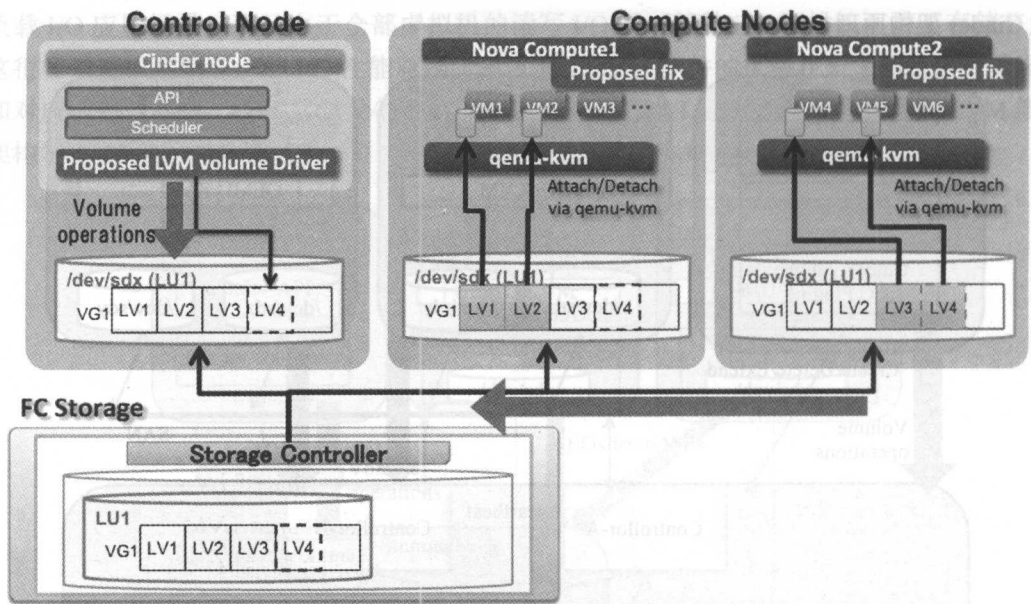


图 10-10 基于 Shared LVM 插件的 Cinder 架构

根据 Hitachi 工程师的描述，共享 LVM 插件在实现过程中首先需要存储管理员的介入，即首先由存储管理员通过 SAN 网络或 iSCSI 实现各个计算节点对存储设备的共享，并在存储设备上创建 LUN 和 VG，然后保证各个计算节点可以同时识别到创建的存储设备端上的 LUN 和 VG。在识别到共享 VG 后，将此 VG 设置到 Cinder 配置文件中作为 Volume 创建时使用的资源池 VG，之后用户便可在运行 Cinder-API 的控制节点进行创建或删除 Volume 等操作。根据 Hitachi 工程师提供的 I/O 性能统计数据（见表 10-5），基于共享 LVM 插件实现的 Cinder 块存储服务与原生 FC 插件实现的块存储服务具有近似的 I/O 速率，并且明显要优于基于 Local LVM 插件实现的 Cinder 块存储。

表 10-5 不同存储插件 I/O 性能对比（单位：KB/s）^①

	I/O size	Local LVM Plugin (单位：KB/s)	SharedLVM Plugin (单位：KB/s)	Raw FC Plugin (单位：KB/s)
Read	1M	49 788.0	85 146.0	88 203.0
	512K	62 382.0	160 517.0	157 810.0
	4K	4026.4	8630.8	8865.2
Write	1M	76 691.0	141 315.0	144 173.0
	512K	59 200.0	142 408.0	144 006.0
	4K	3870.1	7738.9	7867.1

① 数据来源：<https://wiki.openstack.org/wiki/Cinder/NewLVMbasedDriverForSharedStorageInCinder>

(续)

	I/O size	Local LVM Plugin (单位: KB/s)	SharedLVM Plugin (单位: KB/s)	Raw FC Plugin (单位: KB/s)
RandRead	1M	16 152.0	17 665.0	17 105.0
	512K	13 677.0	20 397.0	19 971.0
	4K	417.3	480.6	476.8
RandWrite	1M	15 606.0	17 067.0	16 526.0
	512K	13 666.0	20 381.0	19 955.0
	4K	417.1	4803.3	476.5

表 10-5 中所示的数据为 Hitachi 根据特定的某个实例到存储设备上某块 Volume 的直接 I/O 的数据值。可以看到在不同的 I/O 块和随机读写测试中, Shared LVM 插件与 Raw FC 插件的 I/O 速率比较接近, 并且要优于 Local LVM 插件。从 Shared LVM 插件的设计原理上可以看到, 其工作范畴介于 Local LVM 插件和特定 Vendor 的 FC 插件之间, 并主要应用到厂商没有提供特定 FC 插件的小众或老旧存储设备上, Shared LVM 插件尽管在设计思路上有了一定的可取性, 但是社区并没有真正接受此插件, 具体原因可以参见本节给出的链接。此处介绍此插件的原因, 主要在于很多用户仍然在使用比较老旧或被存储厂商淘汰的存储设备。而由于各种商业原因, 相应的存储设备厂商并没有在 Cinder 项目中实现对这些老旧存储设备的 FC 插件的支持。如果用户仍然希望使用这类存储设备实现 Cinder 块存储服务, 而默认的 LVM 插件在 I/O 性能上又不能满足需求, 则可以参考实现本节介绍的 Shared LVM 插件。

10.2.4 Cinder LVM 插件实现

LVM 插件为 Cinder 默认插件, 在存储节点上部署 Cinder-volume 之后, Cinder-volume 默认便会加载 LVMVolumeDriver 驱动插件。为了便于独立分析, 本节将 Cinder-volume 独立部署在存储节点上。将 Cinder-api 和 Cinder-scheduler 部署在控制节点上。需要指出的是在 OpenStack 集群中, 存储节点是相对的, 任意部署有 Cinder-volume 服务的节点都可以称为存储节点。因此如果将 Cinder-volume 部署在控制节点上, 则控制节点将兼任存储节点。由于 Cinder-volume 负责加载和管理后端存储驱动插件, 并负责进行存储 I/O 处理, 因此在条件允许的情况下尽量将 Cinder-volume 部署在具有较高磁盘 I/O 的物理服务器上。在本例中, 要实现基于 LVM 插件的 Cinder 块存储服务, 只需配置和部署控制节点与存储节点即可。

1. 控制节点安装配置

1) 为 Cinder 项目创建后端数据库。

```
MariaDB [(none)]> CREATE DATABASE cinder;
```

```
MariaDB [(none)]> GRANT ALL PRIVILEGES ON cinder.* TO 'cinder'@'localhost'
IDENTIFIED BY 'CINDER_DBPASS';
MariaDB [(none)]> GRANT ALL PRIVILEGES ON cinder.* TO 'cinder'@'%' IDENTIFIED BY\
'CINDER_DBPASS';
```

2) 创建 Cinder 用户并为其添加 admin 角色。

```
[root@controller1 ~]# openstack user create --password-prompt cinder
[root@controller1 ~]# openstack role add --project service --user cinder admin
```

3) 创建 Cinder 服务项目。

```
[root@controller1 ~]# openstack service create --name cinder --description "OpenStack
Block Storage" volume
[root@controller1 ~]# openstack service create --namecinderv2 --description
"OpenStack Block Storage" volumev2
```

4) 创建 Cinder 服务项目的 API Endpoint。

```
[root@controller1 ~]# openstack endpoint create --publicurl\
http://controller:8776/v2/%(tenant_id)s --internalurl\
http://controller:8776/v2/%(tenant_id)s --adminurl\
http://controller:8776/v2/%(tenant_id)s--region RegionOne volume
[root@controller1 ~]# openstack endpoint create --publicurl\
http://controller:8776/v2/%(tenant_id)s --internalurl\
http://controller:8776/v2/%(tenant_id)s --adminurl\
http://controller:8776/v2/%(tenant_id)s --region RegionOne volumev2
```

5) 安装 Cinder 软件包。

```
[root@controller1 ~]#yum install openstack-cinder python-cinderclient python-oslo-db
```

6) 配置 /etc/cinder/cinder.conf。

```
[DEFAULT]
logdir = /var/log/cinder
state_path = /var/lib/cinder
lock_path = /var/lib/cinder/tmp
volumes_dir = /etc/cinder/volumes
iscsi_helper = lioadm
rootwrap_config = /etc/cinder/rootwrap.conf
auth_strategy = keystone
rpc_backend = rabbit
my_ip = 192.168.142.41

[database]
connection = mysql+pymysql://cinder:CINDER_DBPASS@controller1/cinder

[keystone_authtoken]
auth_host = 127.0.0.1
auth_port = 35357
auth_protocol = http
```



```
auth_uri = http://controller1:5000
auth_url = http://controller1:35357
memcached_servers = controller1:11211
auth_type = password
project_domain_name = default
user_domain_name = default
project_name = service
username = cinder
password = cinder
```

```
[oslo_messaging_rabbit]
rabbit_host = controller1
rabbit_userid = openstack
rabbit_password = openstack
```

```
[oslo_concurrency]
lock_path = /var/lock/cinder
```

7) 启动 Cinder-api 和 Cinder-scheduler 服务。

```
[root@controller1 ~]#systemctl enable openstack-cinder-api.service&& systemctl enable\
openstack-cinder-scheduler.service
[root@controller1 ~]#systemctl start openstack-cinder-api.service&& systemctl start\
openstack-cinder-scheduler.service
```

2. 存储节点安装配置

1) 安装 LVM 管理软件包, 如果需要使用 non-raw 镜像, 则安装 qemu。

```
[root@storage1 ~]#yum install qemu lvm2
```

2) 启动 LVM 元数据服务。

```
[root@storage1 ~]#systemctl enable lvm2-lvm2metad.service
[root@storage1 ~]#systemctl start lvm2-lvm2metad.service
```

3) 创建 PV 和 VG。

```
[root@storage1 ~]#pvcreate /dev/sdb /dev/sdc
//VG创建时候指定的VG名称要与cinder.conf指定的一致
[root@storage1 ~]#vgcreate cinder-volumes /dev/sdb /dev/sdc
```

4) 配置 /etc/lvm/lvm.conf 文件, 使得 LVM 仅扫描 cinder-volumes 卷组的成员磁盘。

```
devices {
...
filter = [ "a/sdb/", "a/sdc","r/*.*"]
...
}
```

5) 安装 Cinder 与 Targetcli 软件包。

```
[root@storage1 ~]#yum install openstack-cinder targetcli python-oslo-db python-
oslo-log MySQL-python
```

6) 配置 /etc/cinder/cinder.conf 文件。

```

[DEFAULT]
rpc_backend = rabbit
auth_strategy = keystone
my_ip = 192.168.142.46
verbose = True
debug = True
glance_api_servers = http://controller1:9292
enabled_backends = lvm           //后端名称为lvm,每个后端都要指定相应的插件驱动来实现
[database]
connection = mysql+pymysql://cinder:CINDER_DBPASS@controller1/cinder
[keystone_authtoken]
auth_uri = http://controller1:5000
auth_url = http://controller1:35357
memcached_servers = controller1:11211
auth_type = password
project_domain_name = default
user_domain_name = default
project_name = service
username = cinder
password = cinder
[matchmaker_redis]
[oslo_concurrency]
lock_path = /var/lock/cinder/tmp
[oslo_messaging_amqp]
[oslo_messaging_notifications]
[oslo_messaging_rabbit]
rabbit_host = controller1
rabbit_userid = openstack
rabbit_password = openstack
//lvm后端实现配置
[lvm]
volume_driver = cinder.volume.drivers.lvm.LVMVolumeDriver //后端驱动
volume_group = cinder-volumes //volume操作所使用的VG
iscsi_protocol = iscsi //存储客户端访问协议
iscsi_helper = lioadm //采用LIO Target

```

7) 启动 Cinder-volume 服务。

```

[root@storagel ~]#systemctl enable openstack-cinder-volume.service target.service
[root@storagel ~]#systemctl start openstack-cinder-volume.service target.service

```

至此，基于 LVM 插件的 Cinder 块存储服务已经配置完成，其配置的关键在于为 Cinder-volume 指定所使用的驱动插件。在 /etc/cinder/cinder.conf 配置文件中，用户可以自定义存储后端（Storage Backup-End），并且可以自定义多个存储后端，而每个后端可以采用不同的驱动来实现。本例中的存储后端为 lvm，其采用的驱动为 LVMVolumeDriver。存储后端的定义与驱动的设置如下：

```
[DEFAULT]
enabled_backends = lvm
...
[lvm]
volume_driver = cinder.volume.drivers.lvm.LVMVolumeDriver
...
```

在控制节点，加载 admin 权限后，可以查看 Cinder 服务的运行情况：

```
[root@controller1 ~]# source adminrc
[root@controller1 ~]# cinder service-list
```

Binary	Host	Zone	Status	State	Updated_at
cinder-scheduler	controller1	nova	enabled	up	2016-10-15T04:23:26.000000
cinder-volume	storage1@lvm	nova	enabled	up	2016-10-15T04:23:24.000000

确认服务正常运行后，在控制节点进行 Volume 创建，如下命令创建了一个 2GB 的 Volume：

```
[root@controller1 ~]# cinder create --name volume1 2
```

如下命令从 image 创建一个大小为 2GB，并且是可引导的 Volume (Bootable Volume)：

```
[root@controller1 ~]# cinder create --name volume2 --image=cirros-0.3.4-x86_64 2
```

查看已经创建的 Volume：

```
[root@controller1 ~]# cinder list
```

ID	Status	Name	Size	Volume Type
3fbalf95-a830-4793-b0bb-7ae85e308822	available	volume1	2	-
8e0e4b0a-e99c-44b4-82c6-4a99b78e8624	available	volume2	2	-

将不可引导的 Volume1 挂载到实例 dvr_ha_server1 上，挂载后实例自动设置磁盘编号：

```
[root@controller1 ~]# nova volume-attach dvr_ha_server1 3fbalf95-a830-4793-b0bb-7ae85e308822 auto
```

Property	Value
device	/dev/vdb
id	3fba1f95-a830-4793-b0bb-7ae85e308822
serverId	78f61e84-f65d-46d4-9d9d-091965f1cac5
volumeId	3fba1f95-a830-4793-b0bb-7ae85e308822

检查 Volume1 是否已经挂载到 dvr_ha_server1 (ID 为 78f61e84-f65d-46d4-9d9d-091965f1cac5):

```
[root@controller1 ~]# cinder list
```

ID	Status	Name	Size	Volume Type
3fba1f95-a830-4793-b0bb-7ae85e308822	in-use	volume1	2	-
8e0e4b0a-e99c-44b4-82c6-4a99b78e8624	available	volume2	2	-

10.2.5 Cinder NFS 插件实现

Cinder 默认使用的存储后端驱动是 LVM, 而 LVM 是一种基于块存储实现的 Cinder 插件, 在基于软件实现的插件中, 除了 LVM 外, 还有很多基于文件系统的插件, 如 Gluster FS、GPFS 和 NFS 等插件。其中, NFS 是一种开源实现的文件系统, 其配置实现和使用非常简单, 本节主要介绍基于 NFS 插件实现的 Cinder 块存储服务, 其他基于文件系统的插件实现在配置上具有相似性。从本质上而言, 基于 NFS 的 Cinder 块存储服务并不允许虚拟机像 LVM 驱动一样访问块级别的存储设备。相反, NFS 驱动在 NFS 共享目录上创建对应 Volume 的文件, 并通过计算节点将文件映射为块设备供虚拟机使用。基于文件系统的块存储服务通常采用控制流与数据流分离的设计, 即用户通过控制节点对运行 Cinder-volume 的存储节点进行 Volume 相关的控制操作, 而虚拟机对存储设备的访问并不经过存储节点, 而是通过计算节点后直接进入 Volume Provider, 这里 Volume Provider 就是 NFS 服务器。要实现基于 NFS 插件的 Cinder 块存储服务, 首先需要配置 NFS 服务器, 并导出共享目录, 具体的配置实现过程如下:

1) 在 NFS 服务器上安装 NFS 软件并启动 NFS 服务。

```
[root@controller1 ~]# yum -y install rpcbind nfs-utils
```

```
[root@controller1 ~]# systemctl enable nfs-server&&systemctl start nfs-server
```

2) 在 NFS 服务器上创建共享目录, 并将其导出。

```
[root@NFSServer ~]# mkdir -p /storage
[root@NFSServer ~]# echo "/storage *(rw, sync, no_root_squash, no_all_squash)" >> /etc/exports
[root@NFSServer ~]# exportfs -a
[root@NFSServer ~]# exportfs
/storage          <world>
```

3) 在存储节点上准备 NFS 服务器的导出目录文本文件, 文件中可以包含多个 NFS 服务器及其导出目录。

```
[root@storagel ~]# touch /etc/cinder/nfsshares
[root@storagel ~]# echo "192.168.142.43:/storage" >> /etc/cinder/nfsshares
[root@storagel ~]# chown root:cinder /etc/cinder/nfsshares
[root@storagel ~]# chmod 0640 /etc/cinder/nfsshares
[root@storagel ~]# more /etc/cinder/nfsshares
192.168.142.43:/storage
```

4) 创建 NFS 客户端挂载点 (也可以使用默认的 \$state_path/mnt)。

```
[root@storagel ~]# mkdir -p /var/lib/cinder/nfs/
[root@storagel ~]# ls -l /var/lib/cinder/nfs/
total 0
[root@storagel ~]# chown -R cinder:cinder /var/lib/cinder/nfs
```

5) 测试 NFS 服务器是否正常导出共享目录。

```
[root@storagel ~]# showmount -e 192.168.142.43
Export list for 192.168.142.43:
/storage *
```

6) 配置 /etc/cinder/cinder.conf 文件。

```
[DEFAULT]
rpc_backend = rabbit
auth_strategy = keystone
my_ip = 192.168.142.46
verbose = True
debug = True
glance_api_servers = http://controller1:9292
enabled_backends = nfs //使用NFS存储后端
[database]
connection = mysql+pymysql://cinder:CINDER_DBPASS@controller1/cinder
[keystone_authtoken]
auth_uri = http://controller1:5000
auth_url = http://controller1:35357
memcached_servers = controller1:11211
auth_type = password
project_domain_name = default
user_domain_name = default
project_name = service
```



```

username = cinder
password = cinder
[matchmaker_redis]
[oslo_concurrency]
lock_path = /var/lock/cinder/tmp
[oslo_messaging_amqp]
[oslo_messaging_notifications]
[oslo_messaging_rabbit]
rabbit_host = controller1
rabbit_userid = openstack
rabbit_password = openstack
//NFS存储后端配置
[nfs]
volume_driver=cinder.volume.drivers.nfs.NfsDriver //NFS插件驱动配置
nfs_shares_config=/etc/cinder/nfsshares //NFS服务器共享目录文件
nfs_mount_point_base=/var/lib/cinder/nfs //本地挂载点
volume_backend_name=nfs //NFS后端名称, 注意必须
与DEFAULT字段相同

```

7) 重启 cinder-volume 服务, 并检查 Cinder 服务运行情况。

```

[root@storagel ~]# systemctl restart openstack-cinder-volume.service
[root@controller1 ~]# cinder service-list
+-----+-----+-----+-----+-----+-----+-----+
| Binary | Host | Zone | Status | State | Updated_at |
| Disabled Reason |
+-----+-----+-----+-----+-----+-----+
| cinder-scheduler | controller1 | nova | enabled | up | 2016-10-15T05:50:36.000000 |
| cinder-volume | storagel@nfs | nova | enabled | up | 2016-10-15T05:50:35.000000 |
+-----+-----+-----+-----+-----+-----+

```

8) 检查存储节点上的本地挂载点是否出现新的目录。正常情况下, Cinder-volume 为每个 NFS 共享目录在挂载点中创建一个新的目录, 目录名称为 HASH 值。

```

[root@storagel ~]# cd /var/lib/cinder/nfs
//由于/etc/cinder/nfsshares中只有一条共享目录, 因此挂载点中只有一个空目录
[root@storagel nfs]# ls -l
total 0
drwxr-xr-x 2 root root 6 Oct 15 13:18 470bd35fe3b28cec71651ea2c9ce9486
[root@storagel nfs]# cd 470bd35fe3b28cec71651ea2c9ce9486
[root@storagel 470bd35fe3b28cec71651ea2c9ce9486]# ls -l
total 0

```

至此, 基于 NFS 插件驱动的 Cinder 块存储服务已经配置完成, 其配置的关键在于为 Cinder-volume 指定所使用的驱动插件。在 /etc/cinder/cinder.conf 配置文件中, 用户可以自

定义存储后端 (Storage Backup-End), 并且可以自定义多个存储后端, 而每个存储后端可以采用不同的驱动来实现。本例中的存储后端为 NFS, 其采用的驱动为 NfsDriver, 后端定义与驱动的设置如下:

```
[DEFAULT]
enabled_backends = nfs
...
[lvm]
volume_driver = cinder.volume.drivers.lvm.NfsDriver
...
```

由于 Cinder 屏蔽了不同存储后端的差异, 并对外提供统一的 API 访问, 因此基于 NFS 驱动的 Cinder 块存储服务与基于 LVM 的 Cinder 块存储服务在 Volume 操作上并无差异。在控制节点加载 admin 权限后, 即可创建 Volume。如下程序创建了一个大小为 2GB 的普通 Volume:

```
[root@controller1 ~]# cinder create --name nfs-volume1 2
```

在控制节点上检查 Volume 是否创建成功:

```
[root@controller1 ~]# cinder list
```

```
+-----+-----+-----+-----+-----+
|          ID          | Status | Name      | Size | Volume Type |
+-----+-----+-----+-----+-----+
| Bootable | Attached to |
+-----+-----+-----+-----+
| e6f17095-a119-4590-b3c3-b10125edd3ac | available | nfs-volume1 | 2 | - |
| false | |
+-----+-----+-----+-----+-----+
```

在存储节点上观察挂载点对应的共享目录下是否有新增的文件:

```
//此处的volume-xxx对应着控制节点中的nfs-volume1
[root@storagel ~]# ls -l /var/lib/cinder/nfs/470bd35fe3b28cec71651ea2c9ce9486
total 0
-rw-rw-rw- 1 root root 2147483648 Oct 15 14:06 volume-e6f17095-a119-4590-b3c3-
b10125edd3ac
//文件大小0MB, 说明创建的文件为稀疏文件(nfs_sparsed_volumes=true)
[root@storagel ~]# cd /var/lib/cinder/nfs/470bd35fe3b28cec71651ea2c9ce9486
[root@storagel 470bd35fe3b28cec71651ea2c9ce9486]# du -ms
0
```

将 nfs-volume1 挂载到 dvr_ha_server2 实例上 (ID 为 36cfe4d5-a8bf-4407-8422-fc4f5a5bd944):

```
[root@controller1 ~]# nova volume-attach dvr_ha_server2\ e6f17095-a119-4590-
b3c3-b10125edd3ac auto
```

```

| Property | Value |
+-----+-----+
| device   | /dev/vdb |
| id       | e6f17095-a119-4590-b3c3-b10125edd3ac |
| serverId | 36cfe4d5-a8bf-4407-8422-fc4f5a5bd944 |
| volumeId | e6f17095-a119-4590-b3c3-b10125edd3ac |
+-----+-----+

[root@controller1 ~]# cinder list
+-----+-----+-----+-----+-----+-----+
| ID | Status | Name | Size | Volume Type |
+-----+-----+-----+-----+-----+
| e6f17095-a119-4590-b3c3-b10125edd3ac | in-use | nfs-volume1 | 2 | - |
| false | ...f5a5bd944 |
+-----+-----+-----+-----+-----+

```

在 `dvr_ha_server2` 实例上使用挂载的块存储:

```

//检查实例是否识别到Volume
[root@dvr_ha_server2 ~]# fdisk -l
Disk /dev/vda: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders, total 2097152 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
/dev/vda1   *        16065      2088449      1036192+  83  Linux

Disk /dev/vdb: 2147 MB, 2147483648 bytes
16 heads, 63 sectors/track, 4161 cylinders, total 4194304 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

//对/dev/vdb格式化
[root@dvr_ha_server2 ~]# mkfs -t ext4 /dev/vdb
mke2fs 1.42.2 (27-Mar-2012)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
131072 inodes, 524288 blocks
26214 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=536870912
16 block groups

```

32768 blocks per group, 32768 fragments per group

8192 inodes per group

Superblock backups stored on blocks:

32768, 98304, 163840, 229376, 294912

Allocating group tables: done

Writing inode tables: done

Creating journal (16384 blocks): done

Writing superblocks and filesystem accounting information: done

//对/dev/vdb进行mount操作

```
[root@dvr_ha_server2 ~]# blkid /dev/vdb
```

```
/dev/vdb: UUID="094bcf22-2c71-4c39-bd2d-ab8af2404bd1" TYPE="ext4"
```

```
[root@dvr_ha_server2 ~]# mkdir /mount
```

```
[root@dvr_ha_server2 ~]# mount -t ext4 /dev/vdb /mount
```

```
[root@dvr_ha_server2 ~]# df -h
```

Filesystem	Size	Used	Available	Use%	Mounted on
/dev	242.3M	0	242.3M	0%	/dev
/dev/vda1	23.2M	18.0M	4.0M	82%	/
tmpfs	245.8M	0	245.8M	0%	/dev/shm
tmpfs	200.0K	72.0K	128.0K	36%	/run
/dev/vdb	2.0G	67.0M	1.8G	4%	/mount

```
[root@dvr_ha_server2 ~]# time dd if=/dev/zero of=/mount/test.txt bs=1M count=100
```

```
100+0 records in
```

```
100+0 records out
```

```
real 0m 6.02s
```

```
user 0m 0.00s
```

```
sys 0m 5.23s
```

```
[root@dvr_ha_server2 ~]# du -ms /mount/test.txt
```

```
100 /mount/test.txt
```

检查存储节点挂载点和 NFS 服务器共享目录有何变化:

```
root@storagel ~]# ls -l /var/lib/cinder/nfs/470bd35fe3b28cec71651ea2c9ce9486
```

```
total 201388
```

```
-rw-rw-rw- 1 107 107 2147483648 Oct 15 14:33 volume-e6f17095-a119-4590-b3c3-b10125edd3ac
```

```
[root@storagel ~]# du -ms /var/lib/cinder/nfs/470bd35fe3b28cec71651ea2c9ce9486
```

```
197 /var/lib/cinder/nfs/470bd35fe3b28cec71651ea2c9ce9486
```

```
[root@NFSServer ~]# ls -l /storage
```

```
total 201388
```

```
-rw-rw-rw- 1 107 107 2147483648 Oct 15 14:33 volume-e6f17095-a119-4590-b3c3-b10125edd3ac
```

```
[root@NFSServer ~]# du -ms /storage
```

```
197 /storage
```

可以看到, 对于 Cinder 客户端和 Nova 虚拟机而言, 基于 NFS 和 LVM 插件实现的 Cinder 块存储服务在使用上并没有任何差异, 不同驱动各自底层的差异完全被 Cinder-volume 屏蔽, 用户虚拟机只需对识别到的块存储进行格式化, 并挂载即可使用, 而无须考虑存储后端如何读写数据。

10.2.6 Cinder Multi-Backends 实现

在实际的生产环境中，用户对存储设备的需求是多样化的，针对不同的业务需求和成本考虑，用户可能希望在高 I/O 的应用系统中使用具有大吞吐量的高性能磁盘，如 SSD 或者 FC SAS 盘，这通常也意味着使用成本的增加。而在简单的数据存储和备份等环境下则希望使用容量型的磁盘以通过较低的成本存储海量数据，如 SATA 或 Near-SAS 盘。以在公有云 QingCloud 上创建性能型（见图 10-11）和容量型（见图 10-12）存储盘为例，创建 10 块 1000GB 的性能存储盘每月费用为 10152 元人民币，而相同数量和容量的容量型存储盘每月只需 4608 元人民币，即容量型存储盘的价格不到性能型存储盘的 1/2，但是性能型存储盘的 I/O 吞吐为 128MB/s，而容量型存储盘的 I/O 吞吐仅为 36MB/s，要远远低于性能型存储盘。对于公有云而言，多后端存储实现的优势在于可为用户提供更灵活的块存储使用选择，而对于私有云建设而言，多后端存储的实现允许用户采用不同的厂商存储插件，以集成不同的存储设备或存储解决方案，从而为不同的开发、测试和生产环境分配不同类型的存储设备。

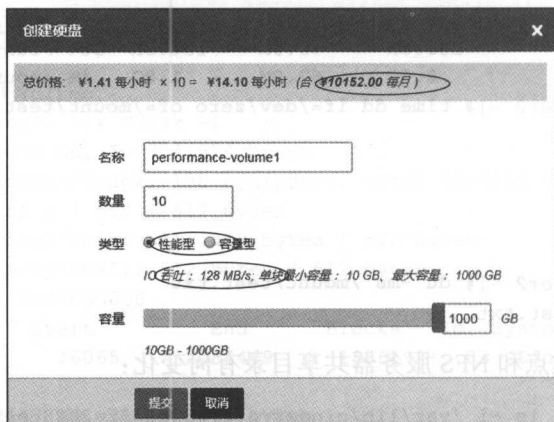


图 10-11 在青云上创建性能型存储盘

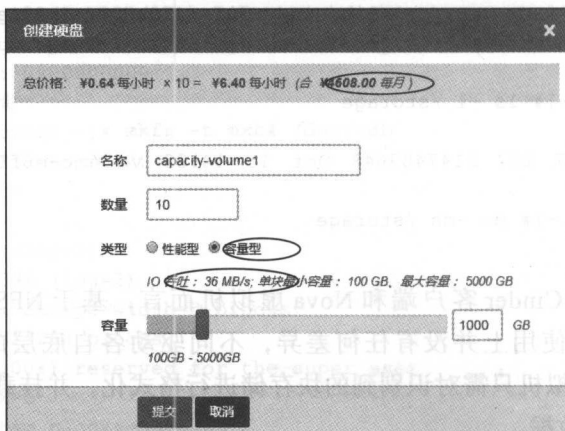


图 10-12 在青云上创建容量型存储盘

在 OpenStack 中, Cinder 项目也实现了多后端存储功能 (Storage Multi-Backends), 通过并行运行多个存储后端插件和驱动。用户可以通过指定存储类型的形式来指定 Volume 应该由哪个运行中的存储后端提供。通过 Cinder 的 Multi-Backends 功能, 用户可以配置多个后端存储解决方案以供相同的 OpenStack 计算资源使用。针对配置的每一个存储后端, Cinder 会启动一个对应的 Cinder-volume 服务 (服务名称格式为 “hostname@backend_name”)。理论上, 用户可以自由组合不同的存储后端插件, 如同时实现 LVM 后端和 Ceph RBD 后端, 或者 IBM 存储后端和 EMC 存储后端。为了简单起见, 本例将对前两节实现的 LVM 和 NFS 驱动后端进行组合, 部署仍然采取控制节点加存储节点模式, 存储多后端插件 (LVM 和 NFS Plugins) 由存储节点加载, 并由控制节点统一管理。Cinder 多后端存储的实现与单独存储后端的实现类似, 不同之处在于, 前者需要在存储节点的 /etc/cinder/cinder.conf 配置文件中同时配置多个存储后端的实现部分, 并在控制节点中为每个存储后端创建对应的存储类型。本例 LVM 和 NFS 多存储后端的存储节点配置文件如下:

```
[DEFAULT]
rpc_backend = rabbit
auth_strategy = keystone
my_ip = 192.168.142.46
verbose = True
debug = True
glance_api_servers = http://controller1:9292
enabled_backends = lvm,nfs                                //同时使用LVM和NFS后端

[database]
connection = mysql+pymysql://cinder:CINDER_DBPASS@controller1/cinder
[keystone_authtoken]
auth_uri = http://controller1:5000
auth_url = http://controller1:35357
memcached_servers = controller1:11211
auth_type = password
project_domain_name = default
user_domain_name = default
project_name = service
username = cinder
password = cinder
[matchmaker_redis]
[oslo_concurrency]
lock_path = /var/lock/cinder/tmp
[oslo_messaging_rabbit]
rabbit_host = controller1
rabbit_userid = openstack
rabbit_password = openstack
//LVM驱动后端配置部分
[lvm]
volume_driver = cinder.volume.drivers.lvm.LVMVolumeDriver
volume_group = cinder-volumes
iscsi_protocol = iscsi
iscsi_helper = lioadm
```

```

volume_backend_name=lvm //存储后端名称
//NFS驱动后端配置部分
[nfs]
volume_driver=cinder.volume.drivers.nfs.NfsDriver
nfs_shares_config=/etc/cinder/nfsshare
nfs_mount_point_base=/var/lib/cinder/nfs
volume_backend_name=nfs //存储后端名称

```

在多后端存储配置中，最关键的部分主要是 `enabled_backends/volume_backend_name` 和 `volume_driver` 的设置。本例中的核心配置段如下：

```

[DEFAULT]
...
enabled_backends = lvm,nfs
...
[lvm]
volume_driver = cinder.volume.drivers.lvm.LVMVolumeDriver
volume_group = cinder-volumes
iscsi_protocol = iscsi
iscsi_helper = lioadm
volume_backend_name=lvm
[nfs]
volume_driver=cinder.volume.drivers.nfs.NfsDriver
nfs_shares_config=/etc/cinder/nfsshare
nfs_mount_point_base=/var/lib/cinder/nfs
volume_backend_name=nfs

```

配置完成后重启存储节点的 `Cinder-volume` 服务，并在控制节点上检查 `Cinder-volume` 存储后端服务的启动情况。正常情况下，每个存储后端会启动一个对应的 `Cinder-volume` 服务。如果用户由单一存储后端变更为多后端，则建议清除 `cinder` 数据库中 `services` 表的内容，然后再重新启动 `Cinder` 服务。本例中 `Cinder` 服务启动完成后控制节点上 `Cinder` 服务运行如下：

```

[root@controller1 ~]# cinder service-list
+-----+-----+-----+-----+-----+-----+-----+
| Binary | Host | Zone | Status | State | Updated_at |
| Disabled Reason |
+-----+-----+-----+-----+-----+-----+
| cinder-scheduler | controller1 | nova | enabled | up | 2016-10-15T07:58:20.000000 |
| cinder-volume | storage1@lvm | nova | enabled | up | 2016-10-15T08:01:07.000000 |
| cinder-volume | storage1@nfs | nova | enabled | up | 2016-10-15T08:01:07.000000 |
+-----+-----+-----+-----+-----+-----+

```

在控制节点上为每个存储后端创建对应的存储类型（`Storage Type`），在多后端场景中进

行 Volume 操作时，需要通过指定存储类型来使用特定的存储后端：

//为LVM后端创建类型lvm

```
[root@controller1 ~]# cinder type-create lvm
```

ID	Name	Description	Is_Public
0d40039d-2831-4950-9aa9-d14e2b9ced8c	lvm	-	True

//为NFS后端创建类型nfs

```
[root@controller1 ~]# cinder type-create nfs
```

ID	Name	Description	Is_Public
47da0d33-c51a-436c-b5d2-495eba3933c8	nfs	-	True

//将类型lvm绑定到名称为lvm的后端

```
[root@controller1 ~]# cinder type-key lvm set volume_backend_name=lvm
```

//将类型nfs绑定到名称为nfs的后端

```
[root@controller1 ~]# cinder type-key nfs set volume_backend_name=nfs
```

```
[root@controller1 ~]# cinder type-list
```

ID	Name	Description	Is_Public
0d40039d-2831-4950-9aa9-d14e2b9ced8c	lvm	-	True
47da0d33-c51a-436c-b5d2-495eba3933c8	nfs	-	True

```
[root@controller1 ~]# cinder extra-specs-list
```

ID	Name	extra_specs
0d40039d-2831-4950-9aa9-d14e2b9ced8c	lvm	{'volume_backend_name': 'lvm'}
47da0d33-c51a-436c-b5d2-495eba3933c8	nfs	{'volume_backend_name': 'nfs'}

现在，便可通过指定不同的 Volume Type 以在不同的存储后端上创建 Volumes：

//在lvm存储后端创建大小为2GB，名称为lvm-volume1的Volume

```
[root@controller1 ~]# cinder create --volume-type lvm --name lvm-volume1 2
```

//在nfs存储后端创建大小为2GB，名称为nfs-volume1的Volume

```
[root@controller1 ~]# cinder create --volume-type nfs --name nfs-volume1 2
```

```
[root@controller1 ~]# cinder list
```

ID	Status	Name	Size	Volume Type
Bootable Attached to				
13adeafc-7279-4c2b-8e3a-a885ebfd4641	available	lvm-volume1	2	lvm
ba309a64-29f6-425f-b79a-640c71dbcecf	available	nfs-volume1	2	nfs

```
| false |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
```

多后端存储配置完成后，在 OpenStack 的 Horizon 项目中通过 Dashboard 界面即可看到创建的 Volume，并且不同的 Volume 具有各自的 Type 属性，如图 10-13 所示。在启用存储多后端后，在 Dashboard 界面创建 Volume 时候也需要指定 Type，如图 10-14 所示。

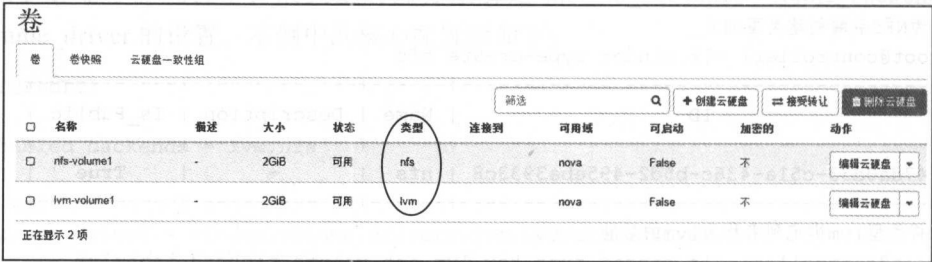


图 10-13 Dashboard 中已创建的存储卷



图 10-14 在 Dashboard 中创建指定类型的存储卷

本例仅演示了基于不同插件和驱动的多存储后端的使用。需要指出的是，不同的存储后端不一定需要使用不同的存储驱动，存储驱动并不是区别存储后端的唯一标准，例如不同的存储后端可以使用相同的 LVM 驱动，但是使用不同的 VG 作为 LVM 的管理对象，如此便可实现相同的驱动管理不同类型的磁盘。如 VG1 完全由 SSD 磁盘组成，VG2 完全由

SATA 磁盘组成, 则可以将 VG1 指定给 LVM-Backend1 后端, VG2 指定给 LVM-Backend2 后端, 从而便可实现类似 QingCloud 一样的性能磁盘和容量磁盘的分组使用。

10.3 Ceph 存储系统

10.3.1 Ceph 背景概述

Ceph 是一种具备卓越性能和高可靠性、高可扩展性的统一、分布式存储系统^①, Ceph 官方网站对其进行抽象介绍的原文是 “Ceph is a unified, distributed storage system designed for excellent performance, reliability and scalability.” 关于为何使用 “Ceph” 这个名称来命名这种分布式文件系统, 相关资料给出的解释是, Ceph 是 Cephalopods 的缩写。Cephalopods 是一类多触手的软体动物, 如章鱼或鱿鱼, 而这类多触手动物是对分布式文件系统一个形象的比喻^②, Ceph 的 Logo 也类似章鱼体型。Ceph 项目起源于其创始人 Sage Weil 在加州大学 Santa Cruz 分校攻读博士期间的研究课题, 而 Ceph 作为开源项目真正为世人所知, 则是因为 Weil 等人 2006 年在 OSDI 杂志发表的论文《Ceph: A Scalable, High-Performance Distributed File System》^③和 Weil 于 2007 年提交的博士论文《CEPH: RELIABLE, SCALABLE, AND HIGH-PERFORMANCE DISTRIBUTED STORAGE》^④。因而 Ceph 开源项目的起源时间普遍认为是 Weil 博士毕业后的 2007 年, 随着云计算尤其是 OpenStack 社区的壮大, Ceph 项目开始广为人知。为了维护 Ceph 项目并提供技术支持, 身为 DeamHost 公司联合创始人和 Ceph 项目创始人的 Sage Weil 于 2011 年成立了 Inktank Storage 公司, 并致力于 Ceph 项目的开发和维护。2014 年 4 月 30 日, RedHat 宣布以 1.75 亿美元收购 Inktank Storage^⑤。RedHat 收购 Inktank Storage 后, Ceph 的整个上游开发团队并入 RedHat, 而 Sage Weil 兼任 RedHat 的首席 Ceph 架构师。Ceph 项目核心代码由 C++ 语言开发, 正如很多开源项目的成长一样, Ceph 也并非如 OpenStack 项目那样在成立之初便吸引业界高度关注, Ceph 在存储业界的兴起和被普遍关注与使用之前已经历了六至七年的发展, 直到 OpenStack 项目引入 Ceph 作为后端存储后, Ceph 社区才凭借 OpenStack 的东风在世界各地迅速成长, 并最终被开源领域的领头羊 RedHat 收购。

Ceph 的两个核心概念便是统一和分布式。“统一” 意味着 Ceph 可以凭借一套存储系统同时向客户提供对象存储、块存储和文件系统存储三种功能, 用户可以对三种功能同时启用, 也可以 “三选一” 使用, 以便在满足不同应用需求的前提下简化部署和运维。而 “分布式” 在 Ceph 系统中则意味着真正的无中心结构和没有理论上限的系统规模可扩展性, 真

① <http://ceph.com/>

② <http://www.ibm.com/developerworks/library/l-ceph/l-ceph-pdf.pdf>

③ <http://ceph.com/papers/weil-ceph-osdi06.pdf>

④ <http://ceph.com/papers/weil-thesis.pdf>

⑤ https://en.wikipedia.org/wiki/Inktank_Storage

正解决了传统 Scale-up 类型存储在扩容过程中所面临的性能瓶颈问题。分布式的 Ceph 存储系统以 Scale-out 的形式可以扩容到 PB 级别，而在实际使用中，Ceph 可以被部署到成千上万台的服务器上，Ceph 的这种分布式部署如图 10-15 所示。作为 Ceph 的最早使用者之一，DreamHost 的对象存储业务在 2013 年已经超过 3PB^①。就目前而言，Ceph 已经成为 OpenStack 社区呼声最高的开源存储解决方案之一，其实际应用主要涉及块存储和对象存储（对应 OpenStack 的 Swift 和 Cinder 项目），并且正逐步向文件系统领域扩展。而随着 2016 年 4 月 Ceph 第七个稳定版本“Jewel”的发布，标志着之前一直处于不稳定阶段的 CephFS（Ceph 文件系统存储）进入稳定版本，因此未来 OpenStack 中的文件系统存储也可被 Ceph 替换。

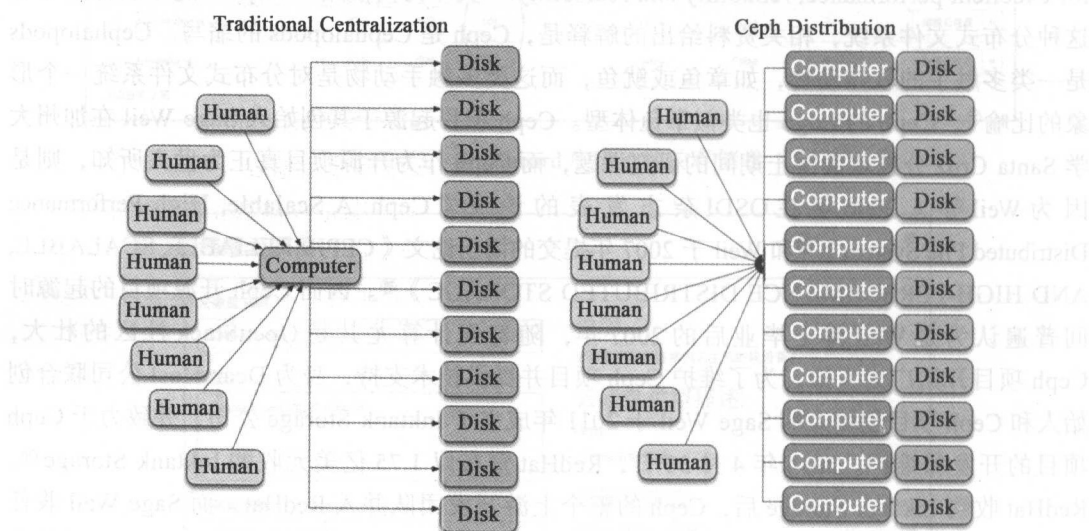


图 10-15 Ceph 分布式与传统中心化对比

Inktank Storage 被 RedHat 收购后，除了 RedHat 外，Ceph 开发的主要参与者还包括 Canonical、CERN、Cisco、Fujitsu、Intel、Red Hat、SanDisk 和 SUSE，而开发语言除 C++ 之外，还有 Python 语言。目前 Ceph 版本发行遵循 LGPL 2.1 协议。Ceph 第一个 LTS 版本于 2013 年 8 月发布，代号为“Dumpling”，版本号为 0.67。Ceph 目前最新版本为 2016 年 4 月发行的“Jewel”，版本号为 10.2.0，这也是 Ceph 发行的第七个稳定版本。Ceph 版本发行历史及现状如表 10-6 所示。从表 10-6 中可以看出，Ceph 版本以 Stable 和 LTS（Long Term Support）类型交替发行，对于生产环境而言，推荐使用长期支持的 LTS 版本，如果想要使用某些已经稳定的新特性，则可以使用 Stable 版本，但是 Stable 版本在使用中发现的 Bug 不一定会在后续得到解决。

^① <http://ceph.com/community/results-from-the-ceph-census/>

表 10-6 Ceph 版本发行历史及现状^①

版本代号	版本号	版本类型	发行时间	预估到期时间	实际到期时间
Dumpling	0.67	LTS	2013-08	2015-03	20150-5
Emperor	0.72	Stable	2013-11		2014-05
Firefly	0.80	LTS	2014-05	20160-01	2015-12
Giant	0.87	Stable	2014-10		2015-04
Hammer	0.94	LTS	2015-04	2016-11	
Infernalis	9.2.0	Stable	2015-11	2016-06	
Jewel	10.2.0	LTS	2016-04	2017-11	

10.3.2 Ceph 架构设计

Ceph 文件系统从设计概念上可以拆分为四个部分，即客户端（Clients）、元数据服务器（Metadata Servers）、对象存储集群（Object Storage Cluster）和集群监控（Cluster Monitor），Ceph 文件系统的概念架构设计如图 10-16 所示。其中，客户端是存储数据的使用者；元数据服务器负责缓存和同步分布式元数据；对象存储集群负责以对象的形式存储全部的用户数据和元数据以及实现其他的关键功能；集群监控则负责实现对整个 Ceph 集群的监控。在实际运行中，Clients 使用 Metadata Server 执行元数据操作（识别和定位数据存储位置），Metadata Server 负责管理已存储数据的位置和即将存储数据的位置，需要注意的是集群元数据也存储在存储集群中（图 10-16 中所示的“Metadata I/O”），而实际的 File I/O 操作发生在 Clients 与 Object Storage Cluster 之间，即 Ceph 文件系统实现了元数据与数据之间的分离，Clients 端先与 Metadata Server 交互并进行元数据操作以获取数据存储位置信息，之后再与 Storage Cluster 交互进行实际的数据存取操作。在这种元数据与数据分离的设计中，高层次的 POSIX 功能，如对文件的打开、关闭和重命名，由 Metadata Servers 进行管理，而 POSIX 功能，如读和写（File I/O），则直接由 Object Storage Cluster 进行管理。

Ceph 与传统文件系统的主要差异就在于，Ceph 并没有将全部的精力集中在文件系统本身，而是分布到 Ceph 文件系统集群中。图 10-17 是 Ceph 文件系统集群的内部简要架构图。Ceph 客户端是 Ceph 文件系统的使用者，Ceph Metadata Daemon 提供了元数据服务，Ceph Object Storage Daemon 提供了实际的存储功能（包括数据和元数据的存储），最后，Ceph Monitor 提供了对整个集群的管理。在实际的 Ceph 文件系统集群中，将会存在大量的 Ceph 客户端，并且有许多的对象存储后端以及多个元数据服务器，同时至少存在一对冗余的监控。

^① <http://docs.ceph.com/docs/master/releases/>

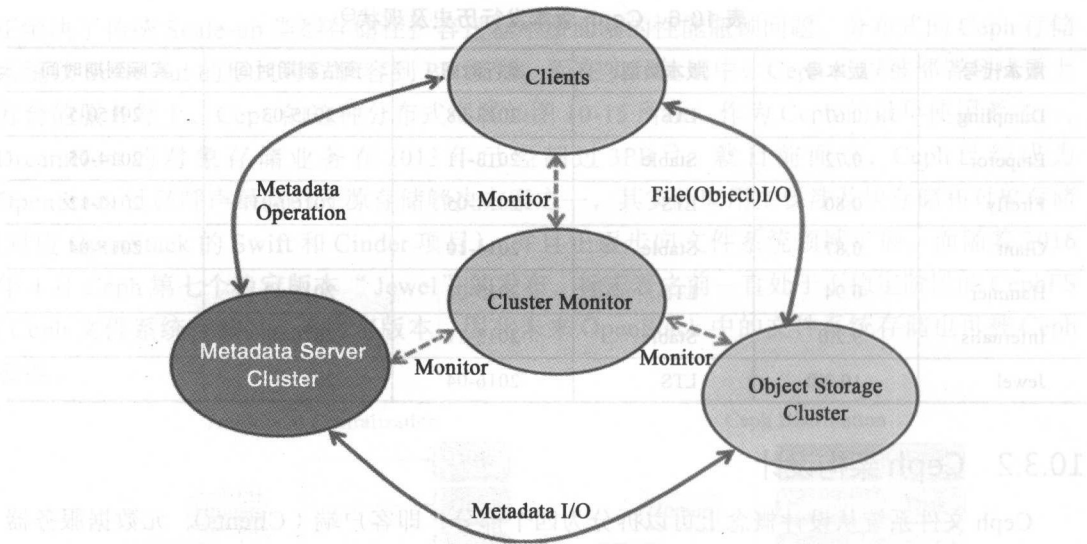


图 10-16 Ceph 文件系统概念架构图

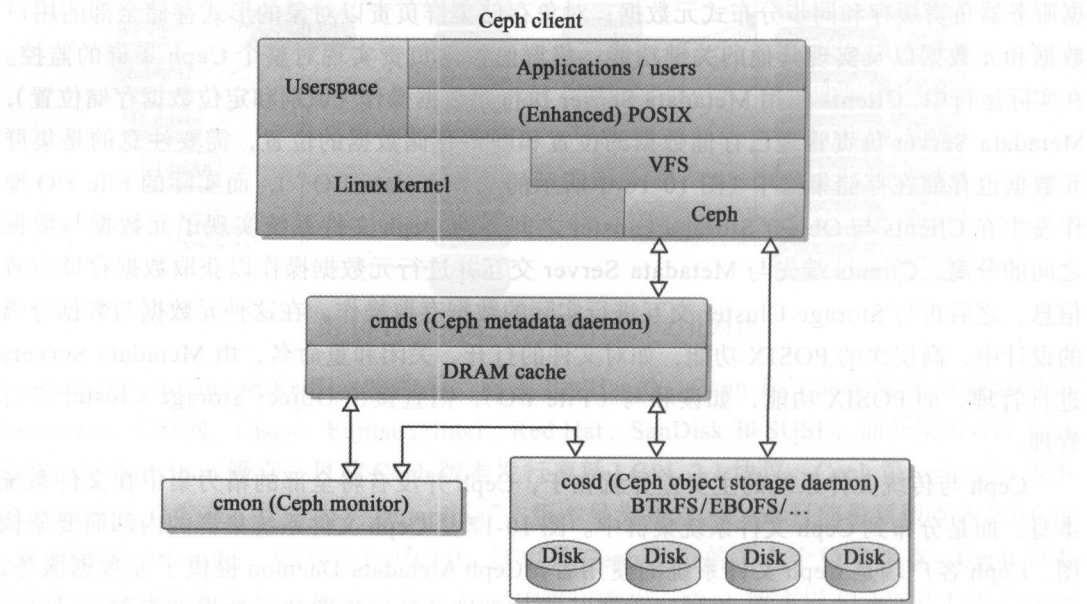


图 10-17 Ceph 内部简要架构

1. Ceph 客户端

早期版本的 Ceph 与 ZFS、GlusterFS 和 Lustre 文件系统类似，使用的是用户空间文件系统（Filesystem in Userspace, FUSE）。使用 FUSE 的好处在于，能够大幅提高开发调试效率，简化为操作系统提供新文件系统所需的工作量。但是，在用户态实现文件系统必然

会引入额外的内核态 / 用户态切换所带来的开销, 对文件系统的性能会产生一定影响。就目前的 Ceph 版本而言, Ceph 文件系统已经集成到 Linux 内核中, 因而可以得到更好的性能。在 Ceph 客户端中, Linux 通过虚拟文件系统 (Virtual File System, VFS) 将通用接口呈现给客户端文件系统, 因此 Ceph 对于文件系统终端用户而言是透明的, 而对于系统管理员而言却略有不同, 因为管理员通常看到的是组成存储系统的大量服务器。从用户角度而言, 仅会看到一个可以执行文件 I/O 的挂载点 (Mount Point), 该挂载点位于一个可读写访问的大型存储系统之上, 用户并不会注意到底层的元数据服务器、集群监控以及组成海量存储池的众多独立存储设备。

与大多数的文件系统类似, Ceph 客户端接口也由 Linux 内核实现, 即所有对文件系统的控制和智能管理操作都由位于内核中的 Ceph 源文件系统自身来实现。但是 Ceph 与常规文件系统的不同之处在于, Ceph 将对文件系统的智能化管理分布到集群中各个节点上, 这种分布式的实现不仅简化了 Ceph 客户端, 同时也为 Ceph 提供了强大的海量存储扩展能力。不同于传统文件系统在文件管理上的分配列表方式, Ceph 采用另外的方式来管理元数据与文件之间的映射关系。从 Linux 角度而言, Ceph 文件系统中每个需要存储的文件都会从元数据服务器获得一个全局唯一的 Inode 号 (Inode Number, INO), 并且对每个文件而言 INO 是其唯一的标志, 在文件被 INO 标志后, Ceph 会根据每个文件的尺寸大小将其转换成不同数目的“对象”, 并结合 Inode 号和对象号 (Object Number, ONO) 为每个对象赋予一个对象 ID (Object ID, OID)。Ceph 对每个对象的 OID 进行简单的 Hash 计算后, 将对象分配到不同的放置组 (Placement Group, PG)。这里的 PG 其实是放置对象的概念性容器, 每个 PG 都通过 PGID 来标志, 而每个 PG 到最终存储对象的底层物理存储设备的映射是一种采用 CRUSH (Controlled Replication Under Scalable Hashing) 算法实现的伪随机映射 (Pseudo-Random Mapping)。通过这种方式, PG 到存储设备的映射并不依赖任何元数据, 而是一个伪随机映射函数, 这种数据映射查找方式最小化了存储负载, 并简化了数据的分布和查找, 而这也是 Ceph 的核心思想之一, 即数据查找时“不用查表, 算算即可”。Ceph 中 PG 的这种映射机制回到整个集群上, 就是集群映射表 (Cluster Map), Cluster Map 是 Ceph 集群中存储设备的最佳表示, 通过 PGID 和 Cluster Map, 用户可以定位任何对象存储位置。

Ceph 客户端包括几个服务接口, 即块设备 (Ceph Block Devices) 接口、对象存储 (Object Storage) 接口和文件系统 (FileSystem) 接口。Ceph 客户端在 Ceph 高层次架构中的位置如图 10-18 所示。其中, 块设备又称 RBD (RADOS Block Devices) 服务, Ceph RBD 提供可调整大小、精简配置并能够进行快照与克隆的块设备, Ceph 的块设备以条带的形式跨越在整个集群中以提高读写性能。Ceph 既支持内核对象, 也支持为避免内核对象过载而直接使用 LIBRBD 的 QEMU Hypervisor。

对象存储又称 RGW (RADOS GateWay)。RGW 服务以 RESTful API 的形式提供兼容 AWS S3 和 OpenStack Swift 的接口。Ceph 文件系统又称 CephFS, CephFS 提供可挂载使用且 POSIX 兼容的文件系统或用户空间文件系统 (FUSE)。CephFS 的内部架构如图 10-19 所示。

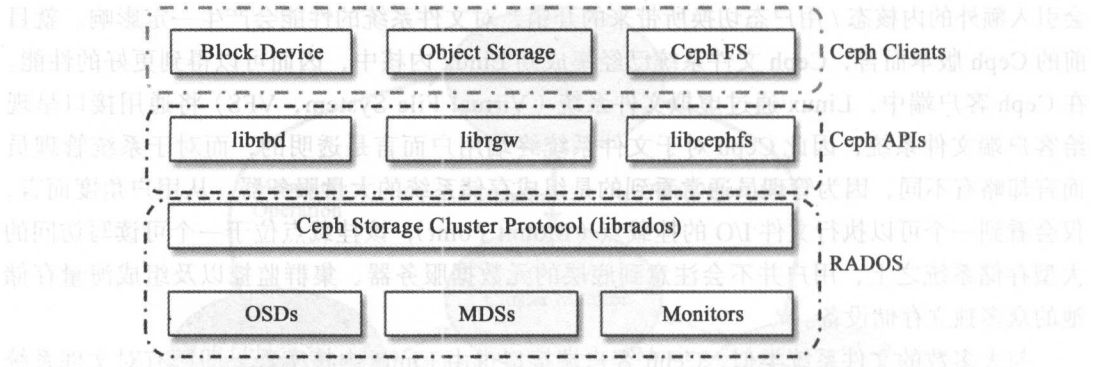


图 10-18 Ceph Clients 在 Ceph 高层次架构中的位置

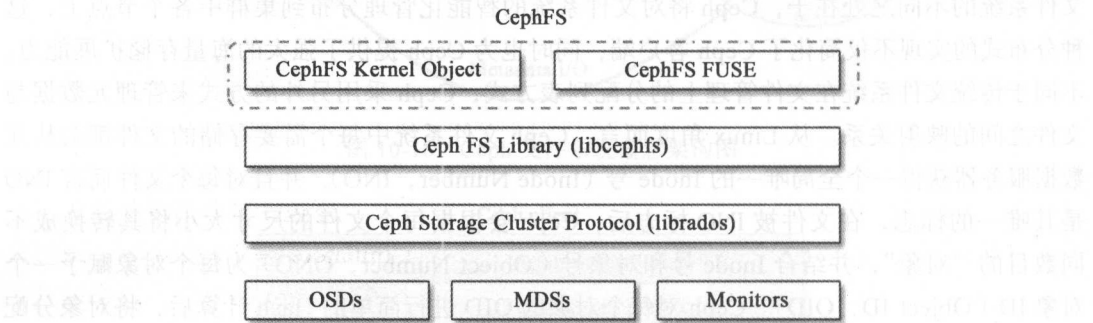


图 10-19 CephFS 内部结构

2. Ceph 元数据服务器

Ceph 元数据服务器 (MetaData Server, MDS) 的主要工作就是管理文件系统的命名空间，虽然 Ceph 集群中的元数据和数据均存储在对象存储集群中，但是为了实现扩展性，二者被分开独立管理。事实上，元数据在可以自适应复制和分发命名空间，以避免热点的元数据服务器集群中被进一步分割。元数据服务器集群对文件系统命名空间的管理如图 10-20 所示，每个 MDS 管理部分命名空间，为了实现冗余和提高性能，各个命名空间部分之间可以重叠。元数据服务器集群与命名空间之间的映射通过动态子树分区 (Dynamic Subtree Partitioning) 实现，这种方式允许 Ceph 适应不断变化的工作负载 (通过在元数据服务器之间迁移命名空间来实现) 并保持局部性能。

在实际应用中，由于每个元数据服务器仅为客户端群体管理命名空间，因此元数据服务器的主要任务便是对缓存元数据的智能管理 (实际的元数据最终将存储到对象存储集群中)。在 Ceph 文件系统中，要写入的元数据首先缓存到临时性日志中，之后再最终写入物理存储。元数据的缓存机制允许 Ceph 将最近写入的元数据以很快的响应速率返回给客户端 (这种使用场景在元数据操作中十分常见)。此外，元数据的日志文件在故障恢复中是否有用？如果元数据服务器故障，则可以通过重现日志文件而恢复元数据并将其存入磁盘。元

数据服务器集群还管理文件系统 Inode 空间，以将文件名转换为元数据信息。通常，元数据服务器将文件名转换为 Ceph 客户端进行文件 I/O 时所需的 Inode、文件尺寸和条带数据。

3. Ceph 监控

Ceph 实现了对集群映射 (Cluster Map) 管理的监控，不过某些故障管理单元的监控由对象存储本身来实现。当对象存储设备出现故障或添加了新设备时，Ceph 监控器会检测并维护 Cluster Map 以使得存储设备出现变化后集群映射仍然有效。Ceph 监控器的这一功能通过分布式的方式来实现。在这种分布式的方式中，Cluster Map 的更新通过现有的通信网络传播到各个节点。此外，Ceph 采用的是具有分布式一致性的 Paxos 算法簇。

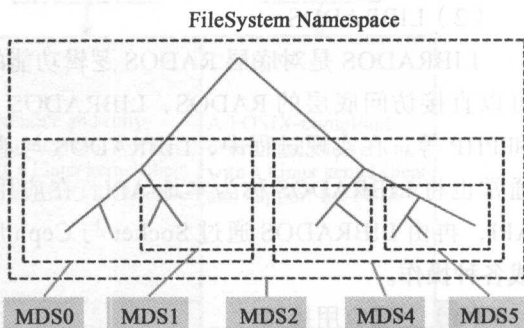


图 10-20 Ceph 元数据服务器命名空间分区

4. Ceph 对象存储

与传统对象存储类似，Ceph 对象存储不仅包括存储功能，还包括对存储对象的智能管理。传统的存储驱动作为一个简单的 Target，仅响应来自 Initiators 端发起的命令，而对象存储设备是一种更为智能的设备，其可以同时充当 Target 和 Initiator 的角色，因此可以支持与其他对象存储设备的通信和合作。从存储角度而言，Ceph 对象存储设备实现的是对象到块的映射（传统意义上，这类任务由位于客户端的文件系统层完成），由于 Ceph 客户端采用的 CRUSH 算法并且无须知道文件在磁盘上的块映射，因此 Ceph 中的底层存储设备可以安全管理和实现存储对象到块的映射。

整体而言，Ceph 抛弃了传统文件系统中心化查表寻址的方式，改而采用计算寻址方式，因而充分发挥了底层存储设备自身的计算能力，从而消除了集群系统对单一中心节点的依赖，并真正实现了集群文件系统的无中心化结构。得益于 Ceph 这种利用存储设备自身计算能力和集群去中心化的设计思想和结构，Ceph 不仅实现了集群文件系统的高可靠性和大容量可扩展性，同时通过分布式计算保证了客户端访问的低延迟和高聚合带宽。Ceph 存储系统的逻辑层次结构如图 10-19 所示，Ceph 存储系统可以分为四个层次，即底层存储系统 (RADOS, Reliable、Autonomic、Distributed Object Store)、底层 API 库 (LIBRADOS)、上层应用接口和外部应用层。

(1) RADOS

RADOS 是具有可靠性、自动化、分布式等特点，并具备自我愈合、自我管理的智能化存储节点集群，因此 RADOS 在本质上是一个具有高可靠性和可用性，并能够无限扩展的物理存储节点集群。RADOS 是 Ceph 存储系统的基础，Ceph 集群的数据最终全部由 RADOS 负责存储和管理，并且数据的复制、恢复和同步等高可用与高可靠的功能也都通过

RADOS 来实现。

(2) LIBRADOS

LIBRADOS 是对底层 RADOS 逻辑功能的抽闲封装 API 库, 上层应用通过 LIBRADOS 可以直接访问底层的 RADOS, LIBRADOS 支持的语言包括 C/C++、Python、Java、Ruby 和 PHP 等。在实现过程中, LIBRADOS 与基于其上所开发的应用位于相同的物理机器上, 通常也将 LIBRADOS 称为本地 API。在应用启动时, 应用首选调用本机上的 LIBRADOS API, 再由 LIBRADOS 通过 Socket 与 Ceph 底层 RADOS 集群中的存储节点进行通信并完成各种操作。

(3) 上层应用接口

Ceph 提供了三种上层应用接口, 即 RADOS GW (RADOS Gateway)、RBD (RADOS Block Device) 和 Ceph FS (Ceph File System)。RADOS GW 对外提供对象存储功能, 并且提供与 Amazon S3 和 OpenStack Swift 兼容的 RESTful API。与 LIBRADOS 相比, RADOS GW 提供的 API 抽象层次更高, 不过功能不如更为接近 RADOS 的 LIBRADOS 丰富。RBD 则对外提供了标准的块设备接口, 通常应用于云计算或虚拟化环境中的虚拟机块设备, 其功能类似于 OpenStack 的 Cinder 项目。Red Hat 目前已经将 RBD 驱动集成到 KVM/QEMU 中以便提高虚拟机访问 Ceph RBD 的性能。Ceph FS 是一个 POSIX 兼容的分布式文件系统, 也是 Ceph 统一存储所提供的三个功能中最晚成熟的功能 (最新的 Jewel 版本才进入稳定阶段), 目前仍然不推荐在生产环境中使用 CephFS。

(4) 应用层

应用层是与 Ceph 存储系统耦合最松的一层, 其主要指使用 Ceph 提供的各种应用接口的应用程序。例如基于 LIBRADOS API 直接开发的对象存储应用, 或者基于 RADOS GW API 开发的对象存储应用以及基于 RBD 实现的云硬盘等。

图 10-21 为 Ceph 功能组件的层次结构图, 图中, LIBRADOS 和 RADOS GW 都提供基于 RADOS 的对象存储 API 接口, 但是二者所能实现的对象存储功能和对开发使用者的要求均不同。RADOS GW 提供的是 RESTfull API, 更多的是面向上层终端用户, 这类用户并不关心底层的 RADOS 实现, 更多的是基于业务逻辑的上层对象存储应用开发, 例如开发公有云上的对象存储服务, 相比使用 LIBRADOS 提供的本地 API, RADOS GW 提供的 RESTful API 并不要求开发者对 Ceph 具备系统全面的理解, 因而采用 RADOS GW 提供的 RESTful API 开发的对象存储应用也很难在效率和性能等方面进行优化和改进。与使用 RADOS GW 接口不同, 使用 LIBRADOS API 要求开发者对底层 RADOS 具有深入了解。LIBRADOS API 向开发者提供了大量的 RADOS 运行状态信息和配置参数, 开发者通过对 LIBRADOS API 的调用, 可以对底层 RADOS 及其存储的对象数据进行状态观察, 并且可对底层 RADOS 系统存储策略实现源自上层的主动控制。因此, 调用 LIBRADOS API 的应用不仅可以操作数据对象, 还可以对底层 RADOS 进行管理和配置, 而这些功能是 RADOS GW 接口所不能实现的。

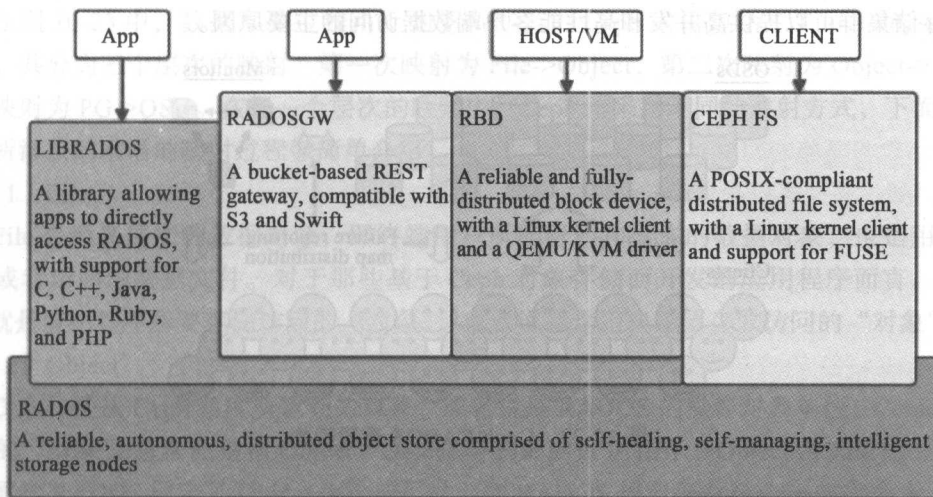


图 10-21 Ceph 逻辑层次结构

10.3.3 Ceph 工作原理

1. RADOS 逻辑架构

Ceph 最为核心的一部分是 RADOS，理解 RADOS 是理解 Ceph 工作原理的基础。RADOS 主要由 Ceph OSD 进程（Object Storage Daemon）和 Ceph Monitor 构成。Ceph 存储集群通常包含众多分布式的 OSD 进程，OSD 进程主要运行在 OSD 节点上，并负责完成数据的存储和维护功能。而 Ceph Monitor 则负责完成集群系统的状态检测和维护功能。在实际运行中，Ceph OSD 和 Monitor 之间相互共享节点状态信息，并通过这些信息计算出集群系统当前运行的整体状态，从而得到一个记录集群全局性系统状态的数据结构，即前面提到的 Cluster Map。对于 Ceph 存储系统而言，Cluster Map 记录了 Ceph 数据对象操作全部所需的关键信息，RADOS 所采用的 CRUSH 算法便是基于 Cluster Map 记录的集群信息进行计算。RADOS 系统的逻辑架构如图 10-22 所示。

当客户端通过 Ceph 提供的各类 API 接口访问 RADOS 时，高并发的客户端程序通过与运行中的 OSD 或 Monitor 进程交互以获取 Ceph 集群的 Cluster Map 信息，然后直接在本地客户端进行计算以获取需要读取或存入数据对象的存储位置。获取对象存取位置后客户端便直接与对应的 OSD 节点交互从而完成对数据对象的读写操作。从客户端与 RADOS 的交互中可以看出，若 Ceph 集群的 Cluster Map 信息保持稳定不变，则客户端对于对象数据的读写访问无须通过元数据服务器进行分配列表的查询即可实现，客户端获取 Cluster Map 信息之后只需进行简单的计算即可获取数据对象存储位置并完成数据访问操作。对于运行中的 RADOS 而言，通常只有在集群 OSD 出现意外故障导致在线 OSD 数目变化或者人为有计划地增删扩容 OSD 节点时，Ceph 集群的 Cluster Map 信息才会出现变化。而对于一个正常运行的生产系统，出现这两种场景的频率显然要远低于客户端数据的读写频率，这也是

Ceph 存储集群可以提供高并发和高性能客户端数据访问的主要原因。

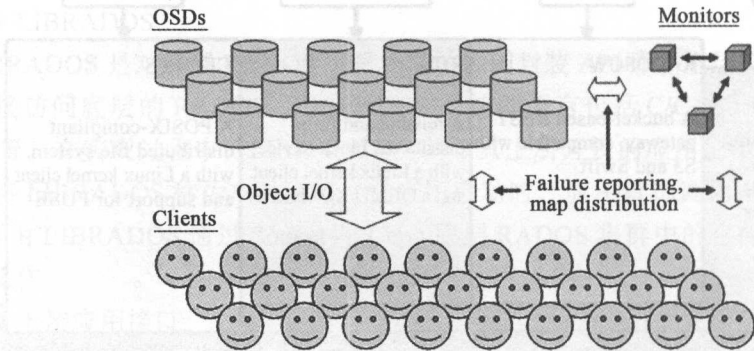


图 10-22 Ceph RADOS 逻辑架构

2. Ceph 数据存储过程

在 Ceph 存储集群中，数据存取的核心基础是 RADOS，因此本节介绍的数据存储过程主要集群 Ceph RADOS，上层应用（如 RBD、RGW 和 CephFS）对于数据对象存储的理论核心均是基于 RADOS 实现的。整体而言，Ceph 存储系统中的数据存储过程大致如下：首先 File 被条带化为数据块，每个数据块均被转换为 Object，在这个过程中，Ceph 根据每个数据块所对应 File 的 Inode Number (INO) 和 Object Number (ONO) 赋予每个 Object 一个 Object ID (OID)；之后 Ceph 利用 Object 的 OID 进行 Hash 计算，并将每个 Object 分配到不同的 PG 中，PG 是逻辑上的 Object 容器，每个 PG 中均可存放多个 Object，此外每个 PG 均有对应的 PGID；最后 Ceph 利用每个 PG 的 PGID 执行 CRUSH 算法以将每个 PG 均映射到物理对象存储设备上，即 OSD 中。这样，逻辑上的数据文件被拆分后以分布式的存储方式存储到 Ceph 集群不同的 OSD 中。文件数据在 Ceph 中的存储过程如图 10-23 所示。

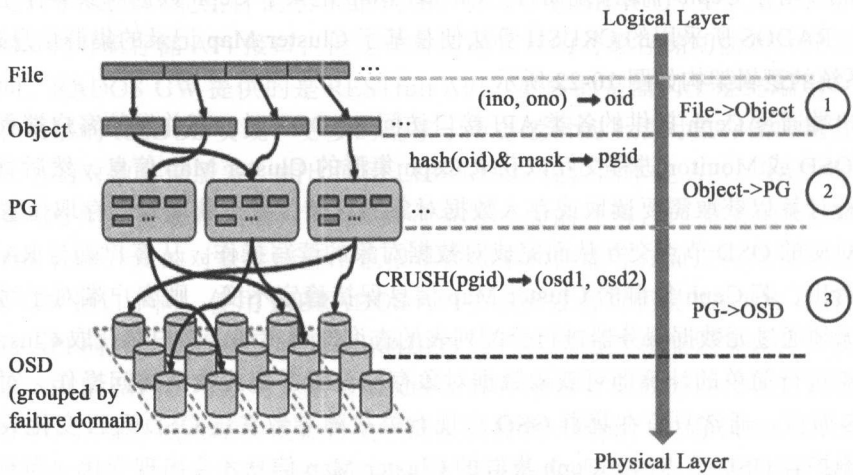


图 10-23 Ceph 文件存储过程

在图 10-23 中, 数据存储过程由上到下所涉及的 Ceph 术语包括了 File、Object、PG、OSD, 共分为三个层次的映射: 第一次映射为 File->Object, 第二次映射为 Object->PG, 第三次映射为 PG->OSD。在每一个层次的映射中, Ceph 均采用不同的映射方式, 下面对数据存储所涉及的术语的映射过程做简单介绍。

(1) File

File 是最高层次的数据对象, 即终端用户所能看到和操作的数据对象, 也是用户需要存储或者访问的数据文件。对于那些基于 Ceph 对象存储而开发的应用程序而言, 这里的 File 就是应用程序所要存储访问的“对象”, 或者说就是用户希望存取访问的“对象”。

(2) Object

Object 是从 Ceph 角度所看到的对象, 或者说是 RADOS 的操作对象单位, Ceph 对象存储中的“对象”通常便是指 Object。Object 与 File 的区别在于, Object 由 File 拆分映射而来, 通常 RADOS 限定了 Object 大小或尺寸, 如 RADOS 规定每个 Object 的大小为 4MB 或 8MB, 以便 Ceph 可以实现对底层存储的组织和管理。所以, 如果 Ceph 客户端向 RADOS 写入较大 Size 的 File, 则对应着会产生很多相同大小的 Object (最后一个 Object 的 Size 可能会小于 RADOS 规定的最大值), 而这些 Object 最终将会被映射到不同的 PG 中。

(3) PG

PG, 即 Placement Group, 是逻辑上的 Object 组织单位或者容器, 其主要作用就是对 Object 的存储进行组织和位置映射。实际应用中, 每个 PG 负责组织若干个 Object, 而 PG 与 Object 之间的映射关系为一对多, 即每个 Object 只能映射到一个 PG 中, 而一个 PG 可以容纳多个 Object。由于 PG 是逻辑上的存储单位, 因此 PG 最终还需映射到 OSD 上, 而 PG 与 OSD 之间为多对多的关系, 即一个 PG 可以映射到多个 OSD, 而一个 OSD 也可以存储多个 PG。在实际应用中, 通常每个 PG 会对应着有一个 Primary OSD 和多个 Secondary OSD, 生产环境中建议至少为 3 个。

(4) OSD

OSD 即是对象存储设备 (Object Storage Device), 是 Ceph 存储集群中的最终物理存储设备, 其主要作用就是存储逻辑上的 PG, 并且通过运行在 OSD 上的 OSD 进程实现不同 OSD 之间的通信以及与 Ceph Monitor 的通信。由于 OSD 是 Ceph 存储集群最终的物理存储设备, Ceph 集群中 OSD 数目的设置很大程度上直接影响到 Ceph 存储集群的配置和性能。理论上, Ceph 集群中的 OSD 数目越多, 则越有利于充分发挥 Ceph 存储系统的高可用、高可靠和高性能等特性。

(5) File 至 Object 的映射

File->Object 的映射是 Ceph 存储系统中数据存储的第一次映射, 也是最简单的映射。这个映射过程可以看成是对 File 进行条带化 (类似磁盘阵列 RAID 上的条带化技术), 条带化后的数据块便是 Object, 而 Object 的大小由 RADOS 规定, 如 2MB 或者 4MB。File->Object 映射所带来的好处之一就是, 将用户提交的 Size 大小不一的 File 拆分为 Size 一致、并可被

RADOS 高效管理的多个 Object。另外一个优势就是将传统对 File 的单机串行处理变为集群多节点下的并行处理。因此对于相同大小的 File，在 Ceph 存储系统中的存取速率要远高于传统文件系统中的存取速率。File 被切分后产生的每个 Object 都将获得唯一的 OID。OID 的生成也非常简单，即简单地对 File 的 INO 和 Object 的 ONO 进行线性组合即是 Object 的 OID。例如，某个名称为 warrior.txt 的 File，其大小为 10MB，INO 为 10。假设 RADOS 规定每个 Object 最大 Size 为 2MB，则 warrior.txt 将被拆分为大小为 2MB 的 5 个 Object，每个 Object 的 OID 分别为 101、102、103、104 和 105。

(6) Object 至 PG 的映射

Object->PG 的映射是 Ceph 存储系统中数据存储的第二次映射，这里的映射采用的是 Hash 算法。Hash 算法的输入参数为 Object 的 OID。图 10-23 中所示的计算公式为 $\text{hash}(\text{oid}) \& \text{mask} \rightarrow \text{pgid}$ ，即首先使用静态哈希函数计算 Object 的 OID 哈希值，以将 OID 映射成近似均匀分布的伪随机值，再将这个伪随机值与给定的 MASK 按位进行“与”（and 或 &）操作，从而找到 PGID，并最终确定该 Object 应该映射到那个 PG。而根据 RADOS 的设计原理，假设 Ceph 集群中的 PG 的总数为 N ，则 MASK 的值取为 $N-1$ （ N 通常为 2 的 m 次方， m 为整数）。因此，图 10-23 中所示的 PGID 计算公司其实是从所有 N 个 PG 中近似均匀地随机选取一个 PG 来存放 Object，这里之所以强调“近似”，是因为根据概率统计的理论，只有样本集足够多才能足够接近随机性，而这里的样本集就是 PG 数目 N 。根据这一原理，只有在存在大量 Object 和 PG 的情况下，Ceph 集群才能够将 Object 近似随机地分布到 PG 中，最终也才能保证存储集群中每个 PG 所组织管理的 Object 数量均衡，而这一点对于 Ceph 存储集群的正常运行和性能提升有很大影响，因为只有数据均匀分布，才不至于某些节点过于负载，某些节点却“无所事事”。为保证 Objects 近似均匀地分布到 PGs 中，一方面需要合理设置 Object 的 Size 以保证 File 被拆分为尽量多的 Object，同时 Ceph 集群中的 PG 数目在 OSD 数目一定的情况下不宜太少，通常 PG 数目需要设置为 OSD 数目的数百倍。至于 Object 的 Size 和 PG 的数目如何精确设置，目前并没有特别严谨的计算公式，更多的需要根据用户的实际使用测试结果来评估。

(7) PG 至 OSD 映射

PG>OSD 的映射是 Ceph 存储系统中数据存储的第三次映射，也是逻辑到物理存储的映射。如图 10-23 所示，RADOS 采用 CRUSH 算法进行 PG 到 OSD 的映射，而 CRUSH 算法的输入参数为需要映射到 OSD 的 PG 所对应的 PGID，CRUSH 算法计算的结果将是一组 m 个 OSD， m 通常也代表了数据对象的副本数，生产系统中建议设置为 3，这样每个 PG 将会映射到 3 个不同的 OSD 中，其中包括 1 个 Primary OSD 和 2 个 Secondary OSD。PG 到 OSD 的映射如图 10-24 所示。

PG 与 OSD 之间的映射采用的是 CRUSH 算法，而 CRUSH 算法依赖 Ceph 集群中的 Cluster Map 和集群数据存储策略（Policy），通常集群策略在配置完成之后就不会发生改变，因此影响 CRUSH 的因素主要是 Cluster Map，而 Cluster Map 会根据集群状态的变化

(如 OSD 意外故障而离线或者人为增删 OSD 数目等)而变化,因此 PG 与 OSD 之间的映射并非固定不变,即特定的 PG 并不会永远固定地映射到特定的某个 OSD。尽管 CRUSH 算法在实际应用中不会经常改变,因此 PG 与 OSD 的映射关系也不会经常变化,但是正因为 CRUSH 算法的动态性,Ceph 集群才具有了在 OSD 等故障情况下的自我愈合能力。例如 OSD 的故障或者增加必然导致 Cluster Map 变化,Cluster Map 的改变必然导致 CRUSH 算法得到的 PG 与 OSD 之间的映射与故障发生之前的映射不一致,这意味着 Ceph 集群会自动将故障 OSD 上的 PG 迁移到正常运行的 OSD 中,通常将这个过程称为数据再平衡(Re-balancing)。OSD 数目增加时,PG 在 OSD 中 Re-balancing 的过程如图 10-25 所示,图中,PG 最初全部分布在 OSD1 和 OSD2 上,由于新增 OSD3 导致 Cluster Map 变化,因此 PG 的映射关系也跟着改变,Re-balancing 过程使得原有的 PG 重新分布到 OSD1、OSD2 和 OSD3 上,而根据 RADOS 的设计,最终的 PG 应该近似均匀地分布到 OSD 集群中。

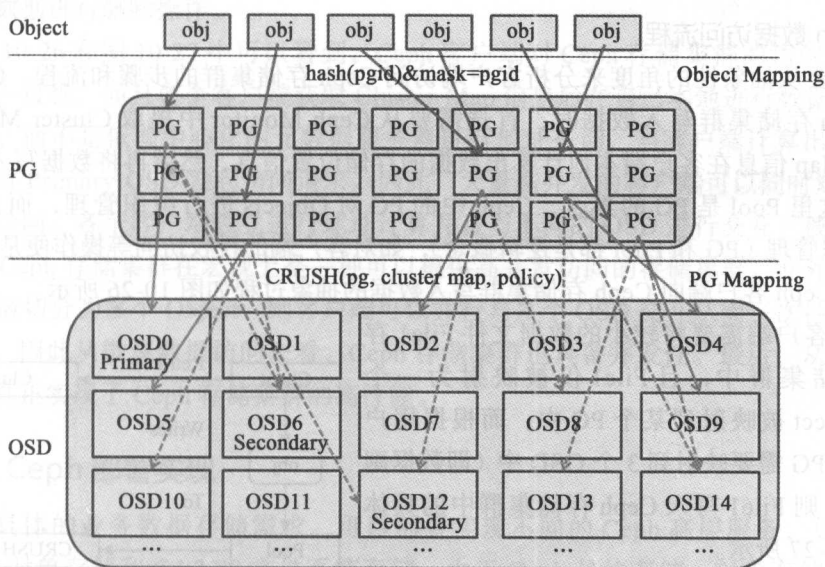


图 10-24 PG 与 OSD 之间的映射

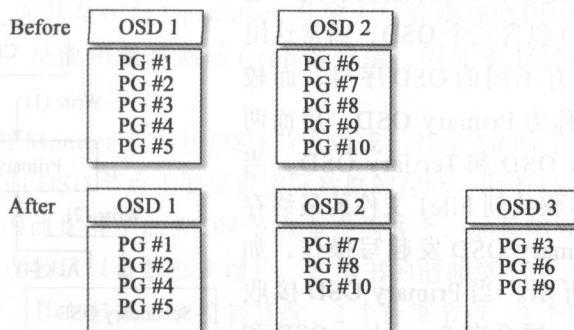


图 10-25 PG Re-balancing 过程

通常, 对于 OSD 故障情况, 数据自动 Re-balancing 是用户所希望的, 但是如果是用户人为扩展 OSD 数目, 则数据 Re-balancing 会消耗大量节点资源, 因此用户并不希望这种场景下出现数据的 Re-balancing, 这也正是使用 CRUSH 算法的主要原因, 即在大规模的 OSD 集群中, 如果 OSD 数目继续增大, Ceph 集群中大多数的 PG 与 OSD 之间的映射关系不会受到影响, 仅有小部分 PG 的映射关系会发生变化而引起数据 Re-balancing。因此, 总结起来, 使用 CRUSH 算法可以使得 PG 与 OSD 之间的映射具有独特的动态性和稳定性, 所以, 除了数据访问无须查表的优势之外, CRUSH 算法的设计也是 Ceph 的核心灵魂。

到此, 文件数据在 Ceph 存储系统中的存储映射过程已经完成, 高层次的文件数据 File 经过 Ceph 内部的三次映射之后, 已经成功分布存储到物理存储设备 OSD 中。从整个存储过程来看, 数据的存取没有经历任何查表操作, 并且第三次映射所采用的 CRUSH 算法为 Ceph 集群的自我愈合和数据稳定性提供了强大支撑。

3. Ceph 数据访问流程

这里主要从抽象性的角度来分析客户端访问 Ceph 存储集群的步骤和流程。Ceph 客户端在向 Ceph 存储集群写入数据时, 首选需要从 Ceph Monitor 中提取 Cluster Map, 并根据 Cluster Map 信息在客户端本地计算出数据的存储位置信息, 然后再将数据写入 Ceph 的 Pools 中。这里 Pool 是 PG 的集合, Ceph 中的 PG 对 Objects 进行组织管理, 而 Pool 又对 PG 进行组织管理 (PG 和 Pool 都是逻辑概念), 如对客户端的授权访问等操作便是通过 Pool 来实现的。Ceph 客户端向 Ceph 存储集群写入数据的抽象过程如图 10-26 所示。

现假设客户端需要将较小的数据文件 File1 存入 Ceph 存储集群中, 且 File1 仅被映射为一个 Object。Object 被映射到某个 PG 中, 而根据用户设置最终的 PG 需要映射到 3 个 OSD 中 (即数据副本为 3 份), 则 File1 写入 Ceph 存储集群中的具体流程如图 10-27 所示。

根据前文的分析, File1 被映射为 Object, Object 被映射到 PG, PG 再通过 CRUSH 算法映射到一组 OSD 中, 这里一组 OSD 包含三个 OSD, 通常这组 OSD 中的三个 OSD 拥有不同的 OSD 序号, 而较为靠前的 OSD 通常被称为 Primary OSD, 后面两个则分别是 Secondary OSD 和 Tertiary OSD。当客户端经过三个映射步骤找到 File1 文件的最终存储位置后, 便会向 Primary OSD 发起写操作, 如图 10-27 中的步骤 1 所示。当 Primary OSD 接收到客户端的写请求后, 便会向 Secondary OSD 和 Tertiary OSD 发起写操作, 如图 10-27 中的步骤 2

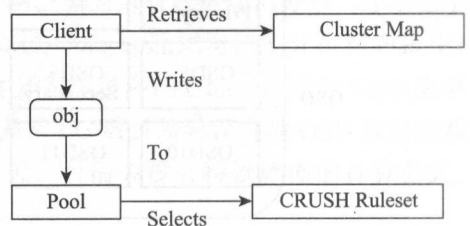


图 10-26 Ceph 客户端写入数据抽象

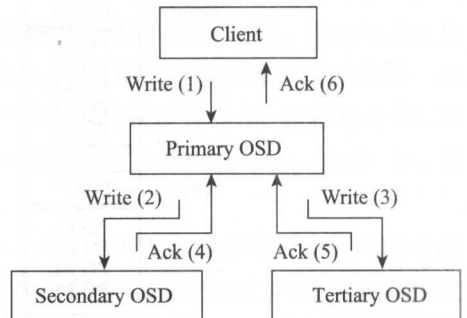


图 10-27 Ceph 数据存入具体流程

和 3 所示。Secondary OSD 和 Tertiary OSD 写操作完成后，便会分别向 Primary OSD 返回写成功应答信号，如图 10-27 中步骤 4 和 5 所示。Primary OSD 接收到 Secondary OSD 和 Tertiary OSD 的写完成信号，并确认自己的写操作也完成后，便向 Ceph 客户端返回本次写操作执行成功的应答信号，如图 10-27 中的步骤 6 所示。在图 10-24 所示的数据写入流程中，客户端只需向 Primary OSD 发出写请求，而不用同时向 Secondary OSD 和 Tertiary OSD 发出写请求，这样的设计缓减了客户端发送多次重复请求所造成的网络带宽压力，在高并发大量客户端同时访问的场景中，这对存储系统的性能提升起到了非常关键的作用。另外，由于客户端需要等待 3 个 OSD 均返回数据写入磁盘成功的信号后才能执行后续的操作，这对于客户端而言将是难以忍受的等待。鉴于此，Ceph 提供了二次应答机制，即一旦全部 OSD 将数据写入缓存，即返回写入成功的信号给客户端，此时客户端可以进行后续操作，而当数据全部写入磁盘后，OSD 再次返回最终的确认信号给客户端，此时客户端便可对之前的数据进行删除操作。

从图 10-26 和图 10-27 中可以看到，Ceph 客户端对 Ceph 存储集群的访问完全无须查表寻址这一过程，而且各个客户端获取 Cluster Map 信息后在自己本地进行数据存储位置的简单计算，而不是依赖中心化的元数据服务器进行地址查询，当客户端计算出数据存储位置后直接对 Primary OSD 发起访问请求。因此，大量高并发的客户端可以同时向 Ceph 存储集群发起访问，各个客户端独立进行地址计算并且与不同的 OSD 进行交互，彼此之间不影响，所以 Ceph 存储集群在宏观上是一种可以提供高并发访问的存储集群。此外，如果客户端的 File 被切分为多个 Object，则客户端可以并行与多个 OSD 交互从而一次性并发获取多个 Object，因此从微观数据访问来看，Ceph 存储集群也具备并发性。最后，从微观到宏观的并发性真正实现了 Ceph 存储集群的高性能。

10.3.4 Ceph 部署实现

根据具体的业务数据存储需求，可以部署实现不同的 Ceph 高层服务，如 RBD 块存储、RGW 对象存储和 CephFS 文件系统存储。由于 Ceph 是块存储、对象存储和文件系统存储的统一存储集群，因此用户也可以基于同一个 Ceph 存储集群实现三种不同的存储方式。在实际使用中，尤其是在与 OpenStack 开源云计算集成使用中，Ceph RADOS Block Device Client (RBD) 是使用最普遍的 Ceph 客户端，因此本节将部署实现 Ceph 集群的 RBD 功能。

Ceph 集群通常由 Monitor 节点和 OSD 节点组成，其中 MON 节点（监控节点）负责整个集群的监控管理，而 OSD 节点主要负责实际数据的存储。由于 Ceph Monitor 集群具有仲裁机制（Quorum），因此集群中的 MON 节点至少为 3 个，如果需要增加 MON 节点数目，则务必保证 MON 节点数目以奇数形式增长。在本书的前面章节中，已经部署了 Mitaka 版本的 OpenStack 集群，其中包括 1 个控制节点、2 个计算节点、2 个网络节点和 1 个存储节点，为了便于后续实验讲解，本章不再新创建集群虚拟机，而是基于原有 OpenStack 集群

节点进行 Ceph 存储集群的部署^①（为了不占用系统资源，将 OpenStack 服务暂时停止）。在规划 Ceph 时，将原有 OpenStack 的 1 个控制节点作为 Ceph 集群的管理和部署节点，将 2 个计算节点和 1 个存储节点作为 MON 和 OSD 节点，剩下的两个网络节点作为 OSD 节点，因此 Ceph 存储集群一共有 3 个 MON 节点和 5 个 OSD 节点，此外还有一个 Admin 节点，Ceph 集群拓扑架构如图 10-28 所示。

在正式部署 Ceph 集群之前，需要确保各个节点之间的时间同步和 Admin 节点对各个 Ceph 节点的 SSH 无密码登录。此外，建议在集群节点中创建 Ceph 部署用户（用户名不能使用 ceph），并且保证 Admin 节点通过此用户可以 SSH 无密码登录到各个 Ceph 节点，并能够在各个 Ceph 节点以 Root 权限执行软件安装配置任务（本节为了便于讲解使用 Root 用户进行 Ceph 部署，生产环境中不推荐使用 Root 用户）。环境准备完成后，可按如下步骤安装部署 Ceph 存储集群（部署使用的 Linux 版本为 CentOS7.2，Ceph 版本为 jewel）。

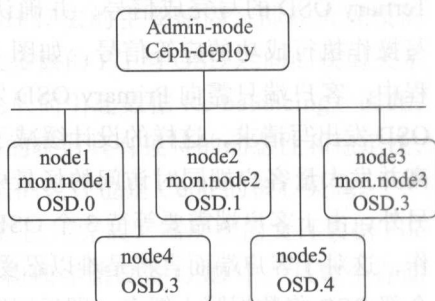


图 10-28 Ceph 存储集群部署节点拓扑

1. Admin 节点准备

Admin 节点上需要准备好 yum 源并安装 Ceph-deploy 部署工具。对于 CentOS/RHEL 系统，需要用到 EPEL 软件包仓库（Extra Packages for Enterprise Linux repository），可以用如下命令安装 EPEL 仓库：

```

yum install -y yum-utils
yum-config-manager --add-repo https://dl.fedoraproject.org/pub/epel/7/x86_64/
yum install --nogpgcheck -y epel-release
rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-7
rm /etc/yum.repos.d/dl.fedoraproject.org*
  
```

此外，还需准备 Ceph 安装包 yum 源，这里安装部署 Ceph 的最新 LTS 版本 Jewel，Ceph 官方网站提供了 Ceph 各个版本的 RPM 安装包^②，用户可以按如下方式手工编辑 yum 仓库文件来实现 Ceph 安装包 yum 源：

```

[ceph-jewel]
name= ceph-jewel
baseurl= https://download.ceph.com/rpm-jewel/el7/noarch/
enabled=1
gpgcheck=1
type=rpm-md
gpgkey=https://download.ceph.com/keys/release.asc
  
```

① 在生产环境中，建议将 Ceph 集群的 MON 节点和 OSD 节点分开部署到专用物理服务器上，物理服务器的配置需求可参考：<http://docs.ceph.com/docs/master/start/hardware-recommendations/>。

② <https://download.ceph.com/>。

yum 源准备完成之后，在 Admin 节点安装 ceph-deploy 部署工具：

```
yum install ceph-deploy
```

Firewalld 和 Selinux 在部署过程中很可能会造成各种诡异情况，如果在私有网络中部署 Ceph，则建议直接关闭 Firewalld 和 Selinux 服务：

```
systemctl stop firewalld.service&& systemctl disable firewalld.service
setenforce 0
```

如果确实要开启防火墙 Firewalld 服务，则开通相应端口：

```
firewall-cmd --zone=public --add-port=6789/tcp --permanent
firewall-cmd --reload
```

在 Admin 节点上创建 ceph-deploy 部署工具专用目录：

```
mkdir my-cluster
cd my-cluster
```

2. 创建 Ceph 集群

在 Admin 节点的 my-cluster 目录，运行 ceph-deploy 命令创建 Ceph 集群，根据部署规划，原 OpenStack 集群的 2 个计算节点和 1 个存储节点作为 Ceph 的 MON 节点，因此 Ceph 集群的创建命令如下：

```
[root@controller1 my-cluster]# ceph-deploy new compute1 compute2 storage1
```

新创建的集群被自动命名为 ceph，集群创建完成后，在 my-cluster 目录中会产生 ceph 配置文件、monitor 密钥文件和 log 文件，具体如下：

```
[root@controller1 my-cluster]# ls -l
total 20
-rw-r--r-- 1 root root 282 Oct 16 12:14 ceph.conf
-rw-r--r-- 1 root root 18948 Oct 16 12:22 ceph.log
-rw----- 1 root root 73 Oct 16 12:14 ceph.mon.keyring
```

ceph.conf 默认配置如下：

```
[root@controller1 my-cluster]# more ceph.conf
[global]
fsid = 1f51609e-e01f-4509-8686-da5f29761c88
mon_initial_members = compute1, compute2, storage1
mon_host = 192.168.142.44,192.168.142.45,192.168.142.46
auth_cluster_required = cephx
auth_service_required = cephx
auth_client_required = cephx
```

Ceph 默认一个 PG 映射到三个 OSD，因此如果需要更改这种映射关系，如仅需要 2 份数据副本，则在 ceph.conf 的 global 配置段添加如下配置：

```
osd pool default size = 2
```

3. 在 Ceph 节点安装 Ceph

在 Admin 节点运行 `ceph-deploy` 命令安装 Ceph, 根据部署规划, 一共部署 5 个 OSD 节点和 3 个 MON 节点, 其中 3 个 MON 节点同时作为 OSD 节点, `ceph-deploy` 的 `install` 命令会在各个 `ceph` 节点上安装 `ceph` 软件包。安装命令如下:

```
ceph-deploy install controller1 compute1 compute2 network1 network2 storage1
```

4. 初始化 Monitors 节点并收集 Keys

在 Admin 节点运行 `ceph-deploy` 的 `mon` 命令以初始化 Ceph 监控 monitors, 命令如下:

```
ceph-deploy mon create-initial
```

初始化完成后, 在 `my-cluster` 中将会新增多个秘钥文件, 具体如下:

```
[root@controller1 my_cluster]# ls -l
total 40
-rw----- 1 root root    71 Oct 16 12:22 ceph.bootstrap-mds.keyring
-rw----- 1 root root    71 Oct 16 12:22 ceph.bootstrap-osd.keyring
-rw----- 1 root root    63 Oct 16 12:22 ceph.client.admin.keyring
-rw-r--r-- 1 root root  282 Oct 16 12:14 ceph.conf
-rw-r--r-- 1 root root 18948 Oct 16 12:22 ceph.log
-rw----- 1 root root    73 Oct 16 12:14 ceph.mon.keyring
```

5. 检查各个 ceph 节点的可用磁盘情况

在 Admin 节点上通过 `ceph-deploy` 工具可以查看远程 Ceph 节点上的磁盘信息, 根据 Ceph 节点输出的磁盘信息选择用于 OSD 创建的物理磁盘, 查看命令如下:

```
[root@controller1 my_cluster]# ceph-deploy disk list compute1 compute2 network1
network2 storage1
.....
[compute2][DEBUG ] /dev/sdb2 other, unknown           //OSD盘
[compute2][DEBUG ] /dev/sdb1 other, unknown           //日志盘
.....
[compute2][DEBUG ] /dev/sdb2 other, unknown           //OSD盘
[compute2][DEBUG ] /dev/sdb1 other, unknown           //日志盘
.....
[compute2][DEBUG ] /dev/sdb2 other, unknown           //OSD盘
[compute2][DEBUG ] /dev/sdb1 other, unknown           //日志盘
.....
[compute2][DEBUG ] /dev/sdb2 other, unknown           //OSD盘
[compute2][DEBUG ] /dev/sdb1 other, unknown           //日志盘
.....
[storage1][DEBUG ] /dev/sdc1 other, unknown           //日志盘
[storage1][DEBUG ] /dev/sdc2 other, unknown           //OSD盘
.....
```

6. 初始化 Ceph 节点 OSD 和日志盘

在 Admin 管理节点使用 `zap` 命令对磁盘初始化。此命令将会删除磁盘上的数据, 运行之前请确认该磁盘无数据存在。`zap` 命令如下:

```
ceph-deploy disk zap {osd-server-name}:{disk-name}
```

本例中对全部 Ceph 节点的磁盘进行初始化, 命令如下:

```
ceph-deploy disk zap compute1:/dev/sdb1 compute1:/dev/sdb2
ceph-deploy disk zap compute2:/dev/sdb1 compute2:/dev/sdb2
ceph-deploy disk zap network1:/dev/sdb1 network1:/dev/sdb2
ceph-deploy disk zap network2:/dev/sdb1 network2:/dev/sdb2
ceph-deploy disk zap storagel:/dev/sdc1 storagel:/dev/sdc2
```

7. 创建 OSD

可以在 Admin 节点通过 ceph-deploy 的 prepare 命令来准备好 OSD, 然后使用 activate 命令激活 OSD, 也可以使用 create 命令一步到位直接创建并激活 OSD。这里使用 prepare 与 activate 命令创建 OSD, 命令如下:

```
ceph-deploy osd prepare {node-name}:{disk}[:{path/to/journal}]
ceph-deploy osd activate {node-name}:{disk}[:{path/to/journal}]
```

本例中共有 5 个 OSD 节点, 每个 OSD 节点上只运行一个 OSD 进程, 每个 OSD 磁盘有一个日志盘, 磁盘 OSD 准备命令如下:

```
ceph-deploy osd prepare compute1:/dev/sdb2:/dev/sdb1
ceph-deploy osd prepare compute2:/dev/sdb2:/dev/sdb1
ceph-deploy osd prepare network1:/dev/sdb2:/dev/sdb1
ceph-deploy osd prepare network2:/dev/sdb2:/dev/sdb1
ceph-deploy osd prepare storagel:/dev/sdc2:/dev/sdc1
```

磁盘准备工作完成后, 激活 (activate) 已经准备好的 OSD, 命令如下:

```
ceph-deploy osd activate compute1:/dev/sdb2:/dev/sdb1
ceph-deploy osd activate compute2:/dev/sdb2:/dev/sdb1
ceph-deploy osd activate network1:/dev/sdb2:/dev/sdb1
ceph-deploy osd activate network2:/dev/sdb2:/dev/sdb1
ceph-deploy osd activate storagel:/dev/sdc2:/dev/sdc1
```

8. 复制 ceph 配置文件和密钥到管理和 ceph 节点

在 Admin 管理节点通过 ceph-deploy 工具即可将配置文件和密钥拷贝到远程 ceph 节点, 命令如下:

```
[root@controller1 my_cluster]# ceph-deploy admin controller1 compute1 compute2
network1 network2 storagel
```

9. 检查 Ceph 集群运行情况

正常情况下, 3 个 Monitor 和 5 个 OSD 进程应该正常运行, 而且集群的状态应该为 HEALTH_OK, 而全部 PG 都应该是 active+clean 状态, 全部 OSD 应该为 up 和 in 状态。由于已经将密钥分发到各个 Ceph 节点, 因此在每个 Ceph 节点上都可以运行如下命令查看集群运行情况:

```
[root@controller1 my_cluster]# ceph -s
cluster eb395961-01f0-4721-9ec1-786b7b0f1304
health HEALTH_OK
monmap e1: 3 mons at {compute1=192.168.142.44:6789/0,compute2=192.168.142.4
5:6789/0,storage1=192.168.142.46:6789/0}, election epoch 6, quorum 0,1,2
compute1,compute2,storage1
osdmap e21: 5 osds: 5 up, 5 in
pgmap v34: 192 pgs, 3 pools, 0 bytes data, 0 objects
176 MB used, 45848 MB / 46024 MB avail
192 active+clean
```

至此，Ceph 存储集群已经配置完成，并正常运行以准备好提供客户端访问。下一节将讨论 Ceph 如何集成到 OpenStack 集群中，以便 OpenStack 的 Nova、Glance 和 Cinder 项目可以直接访问 Ceph 存储集群。

10.4 Ceph 集成 OpenStack

10.4.1 Ceph 集成 OpenStack 概述

OpenStack 作为云计算中的基础架构层 (IaaS)，数据存储是其核心任务之一。在 OpenStack 云计算环境中，数据存储可以分为临时性存储和永久性存储。临时性存储主要由本地文件系统提供，并主要用于 Nova 虚拟机的本地系统和临时数据盘，以及存储 Glance 上传的系统镜像。永久性存储主要由 Cinder 提供的块存储和 Swift 提供的对象存储构成，其中以 Cinder 提供的块存储使用最为广泛，块存储通常以云盘的形式挂载到虚拟机中使用。从 OpenStack 社区用户对存储的需求来看，OpenStack 中需要进行数据存储的三大项目主要是 Nova 项目（虚拟机镜像文件）、Glance 项目（共用模板镜像）和 Cinder 项目（块存储），而且 Nova、Cinder 和 Glance 之间彼此存在数据访问关系，三者在数据存储与访问层面上的关系如图 10-29 所示。

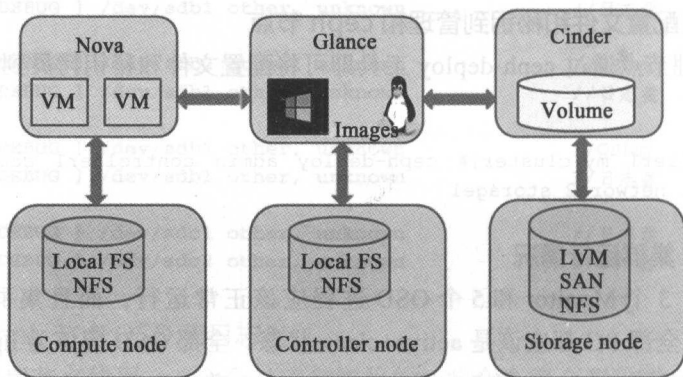


图 10-29 Nova、Cinder 和 Glance 常见数据存储与相互关系

图 10-29 所示中, 由于不同项目的数据分别存储在不同的节点位置, 如 Nova 虚拟机镜像文件通常存储在宿主机本地文件系统中; Glance 镜像文件通常存储在部署 Glance 服务的控制节点本地文件系统中; Cinder 提供的块存储服务将数据存储在不同的存储后端, 常见的有 LVM 后端、NFS 后端和企业级存储 SAN 后端。不同 OpenStack 子项目的数据分开存储很容易造成数据零散混乱、不易管理等缺陷, 这也是作为统一存储的 Ceph 在 OpenStack 社区呼声很高的主要原因。采用 Ceph 统一存储后, OpenStack 的 Nova、Cinder、Glance 和 Swift 项目均可将数据存储到同一个 Ceph 存储集群中。图 10-30 即是 Nova、Cinder 和 Glance 同时将数据存储到 Ceph 集群的示意图。

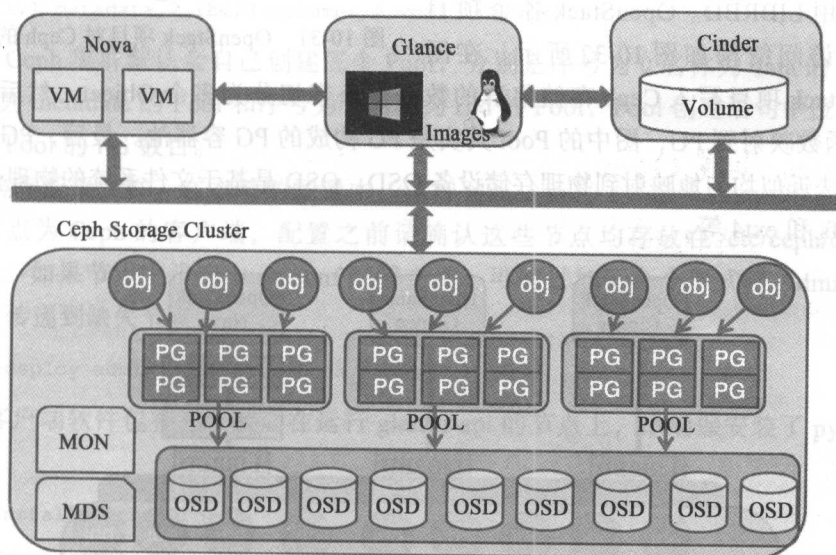


图 10-30 Ceph 作为 Nova、Cinder 和 Glance 数据存储后端

对比图 10-30 和图 10-29 可以看到, 使用 Ceph 作为 OpenStack 的 Nova、Cinder 和 Glance 数据存储后端可以将数据集中存放, 并且能够使 OpenStack 云服务与数据存储池分离, 使得 OpenStack 集群可以专注计算和网络等服务, 而数据存储服务统一交由 Ceph 存储集群来处理。此外, Ceph 还是一个具有高可靠、高可用和高性能等特性的存储集群, 因此采用 Ceph 作为 OpenStack 数据存储后端将极大提升用户数据安全性和访问性能。

Ceph 最初作为科研型开源项目, 成立之初并不如 Swift 等源自商业公司的项目具有较大规模的部署实践, 因此 Ceph 在成立后经历了多年默默无闻的发展, 最终由于其优良的设计加之 OpenStack 社区和 RedHat 等公司对其集成使用, 才成为现在广为人知的开源存储项目。到目前为止, OpenStack 集中进行数据存储的项目包括 Swift、Cinder、Glance 和 Nova, 且均在不同的发行版本上实现了对 Ceph 的支持。OpenStack 各个项目对 Ceph 的支持如图 10-31 所示。

Ceph 与 OpenStack 集成主要用到的是 Ceph 的 RBD 服务，而 Ceph 底层为 RADOS 存储集群，Ceph 通过 LIBRADOS 库来实现对底层 RADOS 的访问。OpenStack 客户端对 Ceph RBD 服务的访问首先要调用 LIBRBD，再由 LIBRBD 通过 LIBRADOS 访问底层的 RADOS。在实际使用中，Nova 需要使用 LibvirtDriver 驱动以通过 Libvirt 和 QEMU 来调用 LIBRBD，而 Cinder 与 Glance 则可以直接调用 LIBRBD。OpenStack 各个项目对 Ceph 的访问结构如图 10-32 所示。在图

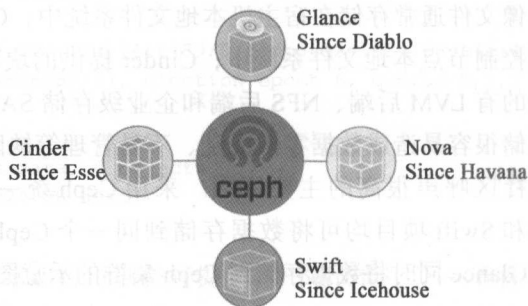


图 10-31 OpenStack 项目对 Ceph 的支持情况

中，OpenStack 项目写入 Ceph 存储集群的数据被条带切分为多个 Object，然后 Object 通过 HASH 函数映射到 PG，图中的 Pool 其实是 PG 构成的 PG 容器池。最后，PG 通过集群 CRUSH 算法近似均匀地映射到物理存储设备 OSD。OSD 是基于文件系统的物理存储设备，如 btrfs、xfs 和 ext4 等。

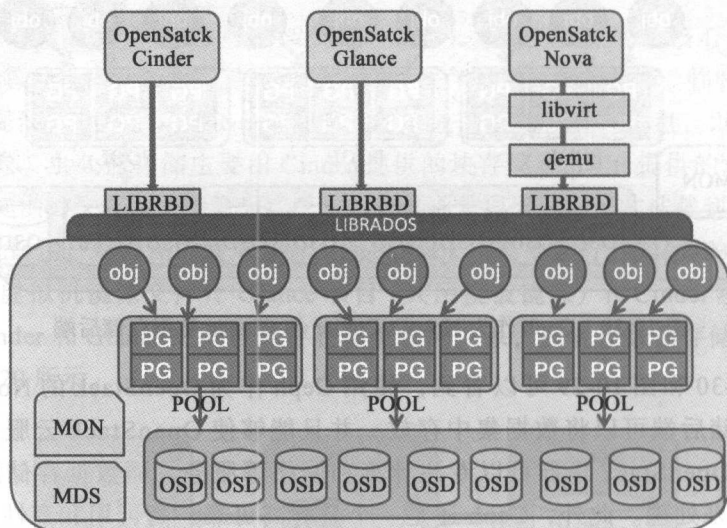


图 10-32 Nova、Cinder 和 Glance 访问 Ceph 集群的结构图

10.4.2 Ceph 集成 OpenStack 准备

在配置 OpenStack 时，使用 Ceph 之前，需要先对 Ceph 进行设置，本节是在假设 Ceph 集群已经如 10.3 节中的结果一样正常运行的基础上展开的。Ceph 默认使用 Pool 的形式存储数据，Pool 实际上是对若干 PG 进行组织管理的逻辑划分，用户可以将不同的数据存入一个 Pool，如 Nova、Cinder 和 Glance 可以使用相同的 Pool，但是这样不便于客户端数据区管理，因此建议为每个客户端创建自己的 Pool。这里将为 Nova、Cinder 和 Glance 分

别创建三个 Pool，Pool 的名称分别为 vms、volumes 和 images。Ceph 集成 OpenStack 的准备工作按如下步骤进行。

1) 为客户端创建 Pool。

```
[root@controller1 ~]# ceph osd pool create volumes 128
pool 'volumes' created
[root@controller1 ~]# ceph osd pool create vms 128
pool 'vms' created
[root@controller1 ~]# ceph osd pool create images 128
pool 'images' created
[root@controller1 ~]# ceph osd lspools
0 data,1 metadata,2 rbd,3 volumes,4 vms,5 images,
```

注意 Ceph 集群默认会自己创建三个 Pool，分别是序号为 0 名称为 data 的 Pool、序号为 1 名称为 metadata 的 Pool 和序号为 2 名称为 rbd 的 Pool，Pool 创建语句中位于最后的数字表示此 Pool 的 PG 数目。

2) 检查配置文件。在本节的配置中，运行 glance-api、cinder-volume 和 nova-compute 服务的节点为 Ceph 的客户端，配置之前请确认这些节点均存放在 /etc/ceph/ceph.conf 配置文件中。如果节点上没有 ceph.conf 配置文件，可通过如下命令形式将 Admin 节点中的 ceph.conf 传递到缺失节点：

```
ceph-deploy admin node_name[node_names]
```

3) 客户端软件包补充安装。在运行 glance-api 的节点上，请确保安装了 python-rbd 软件包。

```
yum install python-rbd
```

在运行 cinder-volume 和 nova-compute 的节点上，确保安装了 ceph-common 软件包。

```
yum install ceph-common
```

4) 授权设置。Ceph 默认会启用 cephx authentication，因此需要为 Nova/Cinder 和 Glance 客户端创建新的用户并授权。这里在 Admin 节点上分别为运行 glance-api 和 cinder-volume 的节点创建 client.glance 和 client.cinder 用户并设置权限：

```
//创建client.cinder并设置权限
```

```
[root@controller1 ~]#ceph auth get-or-create client.cinder mon 'allow r' osd 'allow \
class-read object_prefix rbd_children, allow rwx pool=volumes, allow rwx \
pool=vms, allow rx pool=images'
```

```
[client.cinder]
```

```
key = AQAIlQJY6OVRCBAAHLDej2YV8r9pOoBwsU4hsq==
```

```
//创建client.glance并设置权限
```

```
[root@controller1 ~]#ceph auth get-or-create client.glance mon 'allow r' osd 'allow \
class-read object_prefix rbd_children, allow rwx pool=images'
```

```
[client.glance]
```

```
key = AQA7lQJYKEweJxAA0yfZ+UrWRI8LOsT22tHqjQ==
```

将上述命令为 client.glance 和 client.cinder 用户生成的密码发送到运行 glance-api 和 cinder-volume 的节点上, 本例中 glance-api 运行在控制节点上, cinder-volume 运行在存储节点上:

```
//将client.glance的秘钥传递到glance-api节点
[root@controller1 ~]# ceph auth get-or-create client.glance | tee \
/etc/ceph/ceph.client.glance.keyring
[client.glance]
    key = AQA7lQJYKEweJxAA0yfZ+UrWRI8LOsT22tHqjQ==
[root@controller1 ~]# chown glance:glance /etc/ceph/ceph.client.glance.keyring
//将client.cinder的秘钥传递到cinder-volume节点
[root@controller1 ~]# ceph auth get-or-create client.cinder | ssh storage1 tee \
/etc/ceph/ceph.client.cinder.keyring
[client.cinder]
    key = AQAIlQJY6OVRCBAAHLDej2YV8r9pOoBwsU4hsg==
[root@controller1 ~]# ssh storage1 chown cinder:cinder /etc/ceph/ceph.client.
cinder.keyring
```

运行 nova-compute 服务的节点需要用到 client.cinder 的秘钥文件, 并需将其传递到计算节点:

```
//将client.cinder用户秘钥文件传递到compute1
[root@controller1 ~]# ceph auth get-or-create client.cinder | ssh compute1 tee \
/etc/ceph/ceph.client.cinder.keyring
[client.cinder]
    key = AQAIlQJY6OVRCBAAHLDej2YV8r9pOoBwsU4hsg==
//将client.cinder用户秘钥文件传递到compute2
[root@controller1 ~]# ceph auth get-or-create client.cinder | ssh compute2 tee \
/etc/ceph/ceph.client.cinder.keyring
[client.cinder]
    key = AQAIlQJY6OVRCBAAHLDej2YV8r9pOoBwsU4hsg==
```

nova-compute 节点需要将 client.cinder 用户的秘钥文件存储到 libvirt 中, 当基于 Ceph 后端的 Cinder 卷被 attach 到虚拟机实例时, libvirt 需要用到该秘钥文件以访问 Ceph 集群。在运行 nova-compute 的节点上暂时创建秘钥临时文件:

```
[root@controller1 ~]# ceph auth get-key client.cinder | ssh compute1 tee client.
cinder.key
AQAIlQJY6OVRCBAAHLDej2YV8r9pOoBwsU4hsg==
[root@controller1 ~]# ceph auth get-key client.cinder | ssh compute2 tee client.
cinder.key
AQAIlQJY6OVRCBAAHLDej2YV8r9pOoBwsU4hsg==
```

在运行 nova-compute 的计算节点上将临时秘钥文件添加到 libvirt 中, 然后将其删除:

```
//先生成一个UUID, 全部计算节点可以共同使用此UUID
[root@compute1 ~]# uuidgen
c1b47484-1045-4333-ad7d-55b9354e6f2b
//添加秘钥到compute1的libvirt中
[root@compute1 ~]# cat > secret.xml <<EOF
```

```

<secret ephemeral='no' private='no'>
<uuid> clb47484-1045-4333-ad7d-55b9354e6f2b</uuid>
<usage type='ceph'>
<name>client.cinder secret</name>
</usage>
</secret>
EOF
[root@compute1 ~]# virsh secret-define --file secret.xml
Secret clb47484-1045-4333-ad7d-55b9354e6f2b created
[root@compute1 ~]# virsh secret-set-value --secret clb47484-1045-4333-ad7d-
55b9354e6f2b\
--base64 $(cat client.cinder.key) && rm -rf client.cinder.key secret.xml
//添加秘钥到compute2的libvirt中
[root@compute2 ~]# cat > secret.xml <<EOF
<secret ephemeral='no' private='no'>
<uuid> clb47484-1045-4333-ad7d-55b9354e6f2b</uuid>
<usage type='ceph'>
<name>client.cinder secret</name>
</usage>
</secret>
EOF
[root@compute2 ~]# virsh secret-define --file secret.xml
Secret clb47484-1045-4333-ad7d-55b9354e6f2b created
[root@compute2 ~]# virsh secret-set-value --secret clb47484-1045-4333-ad7d-
55b9354e6f2b\
--base64 $(cat client.cinder.key) && rm -rf client.cinder.key secret.xml

```

这里的 UUID 在后续配置 nova-compute 服务时候还会用到，所以请保存好此 UUID。到此，Ceph 集群已经准备就绪，后面即可配置 Nova、Cinder 和 Glance 客户端以访问 Ceph 集群。

10.4.3 Ceph 集成 Glance

Glance 可以使用不同的后端存储 images，要配置 Glance 默认使用 Ceph RBD 存储镜像，可进行如下操作。

1) Juno 以前的版本按如下方式配置。在 Glance 配置文件 /etc/glance/glance.conf 的 [DEFAULT] 配置段中添加如下配置参数：

```

default_store = rbd
rbd_store_user = glance
rbd_store_pool = images
rbd_store_chunk_size = 8

```

2) Juno 版本按如下方式配置。在 Glance 配置文件 /etc/glance/glance.conf 的 [DEFAULT] 和 [glance_store] 配置段中添加如下配置参数：

```

[DEFAULT]
...

```

```

default_store = rbd
...
[glance_store]
stores = rbd
rbd_store_pool = images
rbd_store_user = glance
rbd_store_ceph_conf = /etc/ceph/ceph.conf
rbd_store_chunk_size = 8

```

3) Juno 以后的版本按如下方式配置。在 Glance 配置文件 `/etc/glance/glance.conf` 的 `[glance_store]` 配置段中添加如下配置参数：

```

[glance_store]
stores = rbd
default_store = rbd
rbd_store_pool = images
rbd_store_user = glance
rbd_store_ceph_conf = /etc/ceph/ceph.conf
rbd_store_chunk_size = 8

```

对于任何版本的 OpenStack，如果想使用 `copy-on-write` 功能，则在 `[DEFAULT]` 配置段中添加如下配置行：

```
show_image_direct_url = True
```

如果 Glance 之前使用默认的本地文件系统存储 images，现在想要更改为使用 Ceph 存储 images，则可以将如下的默认配置：

```

[glance_store]
default_store = file
filesystem_store_datadir = /var/lib/glance/images/

```

更改为：

```

[glance_store]
stores = rbd
default_store = rbd
rbd_store_pool = images
rbd_store_user = glance
rbd_store_ceph_conf = /etc/ceph/ceph.conf
rbd_store_chunk_size = 8

```

本例中配置完成后，完整的 `glance-api.conf` 配置文件如下：

```

[DEFAULT]
show_image_direct_url = True
[database]
connection = mysql+pymysql://glance:GLANCE_DBPASS@controller1/glance
[glance_store]
default_store = rbd
rbd_store_pool = images
rbd_store_user = glance

```



```

rbd_store_ceph_conf = /etc/ceph/ceph.conf
rbd_store_chunk_size = 8
[image_format]
[keystone_authtoken]
auth_uri = http://controller1:5000
auth_url = http://controller1:35357
memcached_servers = controller1:11211
auth_type = password
project_domain_name = default
user_domain_name = default
username = glance
password = glance
project_name = service

```

4) 重启 glance 服务:

```

[root@controller1 ~]# systemctl restart openstack-glance-api.service
[root@controller1 ~]# systemctl restart openstack-glance-registry.service

```

5) 上传镜像验证。下载测试镜像:

```

[root@controller1 ~]# wget -P /tmp/images\
http://download.cirros-cloud.net/0.3.4/cirros-0.3.4-x86_64-disk.img

```

上传镜像, 此时默认应该上传到 Ceph 集群名称为 images 的 Pool 中:

```

[root@controller1 data]# glance image-create --name "cirros-on-ceph" --file\
/data/cirros-0.3.4-x86_64-disk.img --disk-format qcow2 --container-format bare\
--visibility public --progress

```

检查 Ceph 存储集群中名称为 images 的 Pool 是否已经上传有镜像:

```

[root@controller1 data]# rbd ls images
8ca07a37-01c5-4472-be3b-91e0ad0e5c2e

```

可以看到, 镜像已被上传至 Ceph 存储集群中, Glance 成功与 Ceph 集成并默认使用 Ceph RBD 作为其存储镜像的后端存储。

10.4.4 Ceph 集成 Cinder

Cinder 是最早实现对 Ceph 支持的 OpenStack 项目, 本节将介绍如何将 Ceph 集成为 Cinder 块存储服务的存储后端。10.2 节中介绍了 Cinder 的插件式架构, Cinder 支持同时使用多种存储后端, 各个存储后端的实现只需配置相应的插件驱动即可。与配置 LVM 和 NFS 插件驱动类似, Ceph 块存储服务也实现了自己的 RBD 驱动, 因此只需在 Cinder 配置文件中设置相应的 Ceph RBD 驱动即可实现 Cinder 与 Ceph 的集成。

要配置 Cinder 使用 Ceph 后端, 只需在 Cinder 的配置文件 `/etc/cinder/cinder.conf` 中添加 ceph 后端, 并设置 ceph 驱动, 具体如下:

```
[DEFAULT]
```

```
...
enabled_backends = ceph
...
[ceph]
volume_driver = cinder.volume.drivers.rbd.RBDDriver
rbd_pool = volumes
rbd_ceph_conf = /etc/ceph/ceph.conf
rbd_flatten_volume_from_snapshot = false
rbd_max_clone_depth = 5
rbd_store_chunk_size = 4
rados_connect_timeout = -1
glance_api_version = 2
rbd_user = cinder
rbd_secret_uuid = c1b47484-1045-4333-ad7d-55b9354e6f2b //准备阶段生成的UUID
```

如果配置 Cinder 使用多后端, 则 `glance_api_version = 2` 配置行必须位于 `cinder.conf` 文件的 [DEFAULT] 配置段。本例将配置 Cinder 使用 LVMNFS 和 Ceph 后端, 因此该配置行位于 [DEFAULT] 配置段中。本例配置完成后最终的 `cinder.conf` 配置文件内容如下:

```
[root@storage1 ~]# more /etc/cinder/cinder.conf | grep -vE "^$|^#"
[DEFAULT]
rpc_backend = rabbit
auth_strategy = keystone
my_ip = 192.168.142.46
verbose = True
debug = True
glance_api_servers = http://controller1:9292
enabled_backends = lvm,nfs,ceph
glance_api_version = 2
[database]
connection = mysql+pymysql://cinder:CINDER_DBPASS@controller1/cinder
[keystone_authtoken]
auth_uri = http://controller1:5000
auth_url = http://controller1:35357
memcached_servers = controller1:11211
auth_type = password
project_domain_name = default
user_domain_name = default
project_name = service
username = cinder
password = cinder
[oslo_concurrency]
lock_path = /var/lock/cinder/tmp
[oslo_messaging_rabbit]
rabbit_host = controller1
rabbit_userid = openstack
rabbit_password = openstack
[lvm]
volume_driver = cinder.volume.drivers.lvm.LVMVolumeDriver
volume_group = cinder-volumes
```

```

iscsi_protocol = iscsi
iscsi_helper = lioadm
volume_backend_name=lvm
[nfs]
volume_driver=cinder.volume.drivers.nfs.NfsDriver
nfs_shares_config=/etc/cinder/nfsshare
nfs_mount_point_base=/var/lib/cinder/nfs
volume_backend_name=nfs
[ceph]
volume_driver = cinder.volume.drivers.rbd.RBDDriver
rbd_pool = volumes
rbd_ceph_conf = /etc/ceph/ceph.conf
rbd_flatten_volume_from_snapshot = false
rbd_max_clone_depth = 5
rbd_store_chunk_size = 4
rados_connect_timeout = -1
rbd_user = cinder
rbd_secret_uuid = clb47484-1045-4333-ad7d-55b9354e6f2b
volume_backend_name=ceph

```

配置完成后，在存储节点上重启 cinder-volume 服务：

```
systemctl restart openstack-cinder-volume.service
```

在控制节点上检查 cinder-volume 服务启动情况：

```
[root@controller1 ~]# cinder service-list
```

```

+-----+-----+-----+-----+-----+-----+
| Binary | Host | Zone | Status | State | Updated_at |
| Disabled Reason |
+-----+-----+-----+-----+-----+-----+
| cinder-scheduler | controller1 | nova | enabled | up | 2016-10-15T22:21:34.000000 |
| cinder-volume | storage1@ceph | nova | enabled | up | 2016-10-15T22:21:33.000000 |
| cinder-volume | storage1@lvm | nova | enabled | up | 2016-10-15T22:21:34.000000 |
| cinder-volume | storage1@nfs | nova | enabled | up | 2016-10-15T22:21:34.000000 |
+-----+-----+-----+-----+-----+-----+

```

在 OpenStack 控制节点上为 Cinder 的 ceph 存储后端创建对应的 Type：

```
[root@controller1 ~]# cinder type-create ceph
```

```

+-----+-----+-----+-----+
| ID | Name | Description | Is_Public |
+-----+-----+-----+-----+
| 0237e7cf-d0b8-4201-8ea2-79bb8e3c9acd | ceph | - | True |
+-----+-----+-----+-----+

```

```
[root@controller1 ~]# cinder type-list
```

ID	Name	Description	Is_Public
0237e7cf-d0b8-4201-8ea2-79bb8e3c9acd	ceph	-	True
0d40039d-2831-4950-9aa9-d14e2b9ced8c	lvm	-	True
47da0d33-c51a-436c-b5d2-495eba3933c8	nfs	-	True

```
[root@controller1 ~]# cinder type-key ceph set volume_backend_name=ceph
```

```
[root@controller1 ~]# cinder extra-specs-list
```

ID	Name	extra_specs
0237e7cf-d0b8-4201-8ea2-79bb8e3c9acd	ceph	{'volume_backend_name': 'ceph'}
0d40039d-2831-4950-9aa9-d14e2b9ced8c	lvm	{'volume_backend_name': 'lvm'}
47da0d33-c51a-436c-b5d2-495eba3933c8	nfs	{'volume_backend_name': 'nfs'}

使用 Cinder 的 Ceph 存储后端创建 Volumes，在指定 Type 为 Ceph 的情况下，创建的 Volume 应该位于 Ceph 集群中名称为 volumes 的 Pool 中：

```
[root@controller1 ~]# cinder create --volume-type ceph --name ceph-volume1 2
```

```
[root@controller1 ~]# cinder list
```

ID	Status	Name	Size	Volume Type
13adeafc-7279-4c2b-8e3a-a885ebfd4641	available	lvm-volume1	2	lvm
18c6dc44a-6ec3-4187-8de9-5daa1dec0b01	available	ceph-volume1	2	ceph
ba309a64-29f6-425f-b79a-640c71dbcecf	available	nfs-volume1	2	nfs

检查 Ceph 存储集群中名称为 volumes 的 Pool，此时 Pool 中应该存在一个 Volume 中：

```
[root@controller1 ~]# rbd ls volumes
```

```
volume-8c6dc44a-6ec3-4187-8de9-5daa1dec0b01
```

可以看到，Cinder 成功地与 Ceph 存储集群集成，并将 Ceph 作为其多个后端存储之一以向用户提供块存储服务。用户在后续创建 Volume 时，只需指定 Type 为 ceph 即可使用 Ceph 提供的 RBD 服务。

10.4.5 Ceph 集成 Nova

从 Havana 版本开始，OpenStack 的 Nova 项目支持虚拟机从 Ceph RBD 中启动。为了

从 Ceph RBD 中启动虚拟机，必须将 Ceph 配置为 Nova 的临时后端。在 Nova 与 Ceph 集成配置中，推荐在计算节点的 Ceph 配置文件中启用 RBD Cache 功能，并且为了便于故障排查，建议配置 admin socket 参数，每个使用 Ceph RBD 的虚拟机都有一个 socket 将有利于虚拟机性能分析和故障解决。在计算节点 `/etc/ceph/ceph.conf` 配置文件中的 `[client]` 配置段添加如下内容：

```
[client]
    rbd cache = true
    rbd cache writethrough until flush = true
    admin socket = /var/run/ceph/guests/$cluster-$type.$id.$pid.$cctid.asok
    log file = /var/log/qemu/qemu-guest-$pid.log
    rbd concurrent management ops = 20
```

在创建 `ceph` 配置文件中指定路径并设置文件权限，用户 `qemu` 和组 `libvirt` 根据 Linux 发行版本的不同而不同。如果不确定，则直接按照如下方式设置：

```
mkdir -p /var/run/ceph/guests/ /var/log/qemu/
#chown qemu:libvirt /var/run/ceph/guests /var/log/qemu/
chown 777 -R /var/run/ceph/guests /var/log/qemu/
```

1) 如果是 Havana 或 Icehouse 版本，则按如下配置计算节点。在每个计算节点的 `[DEFAULT]` 配置段添加如下内容：

```
libvirt_images_type = rbd
libvirt_images_rbd_pool = vms
libvirt_images_rbd_ceph_conf = /etc/ceph/ceph.conf
libvirt_disk_cachemodes="network=writeback"
rbd_user = cinder
rbd_secret_uuid = clb47484-1045-4333-ad7d-55b9354e6f2b //准备阶段生成的UUID
libvirt_inject_password = false
libvirt_inject_key = false
libvirt_inject_partition = -2
```

如果需要使用 `live-migration` 功能，则按如下方式配置 `libvirt_live_migration_flag` 参数：

```
libvirt_live_migration_flag="VIR_MIGRATE_UNDEFINE_SOURCE,VIR_MIGRATE_PEER2PEER,VIR_MIGRATE_LIVE,VIR_MIGRATE_PERSIST_DEST,VIR_MIGRATE_TUNNELLED"
```

2) 如果是 Juno 版本，则按如下配置。Juno 版本中 RBD 配置参数被移到 `[libvirt]` 配置段，在 `[libvirt]` 配置段中添加如下内容：

```
[libvirt]
images_type = rbd
images_rbd_pool = vms
images_rbd_ceph_conf = /etc/ceph/ceph.conf
rbd_user = cinder
rbd_secret_uuid = clb47484-1045-4333-ad7d-55b9354e6f2b //准备阶段生成的UUID
disk_cachemodes="network=writeback"
inject_password = false
inject_key = false
```



```
inject_partition = -2
live_migration_flag="VIR_MIGRATE_UNDEFINE_SOURCE,VIR_MIGRATE_PEER2PEER,VIR_
MIGRATE_LIVE,VIR_MIGRATE_PERSIST_DEST,VIR_MIGRATE_TUNNELLED"
```

3) 如果是 Kilo 及其以后的版本, 则按照如下配置。Kilo 以后版本支持虚拟机的临时 root 磁盘 discard 功能^①, 其他配置与 Juno 类似, 在 [libvirt] 配置段添加如下内容:

```
[libvirt]
images_type = rbd
images_rbd_pool = vms
images_rbd_ceph_conf = /etc/ceph/ceph.conf
rbd_user = cinder
rbd_secret_uuid = c1b47484-1045-4333-ad7d-55b9354e6f2b    //准备阶段生成的UUID
disk_cachemodes="network=writeback"
inject_password = false
inject_key = false
inject_partition = -2
live_migration_flag="VIR_MIGRATE_UNDEFINE_SOURCE,VIR_MIGRATE_PEER2PEER,VIR_
MIGRATE_LIVE,VIR_MIGRATE_PERSIST_DEST,VIR_MIGRATE_TUNNELLED"
hw_disk_discard = unmap    //root disk discard option
```

本例中计算节点配置完成后, 完整的 nova.conf 配置如下:

```
[root@computel ~]# more /etc/nova/nova.conf|grep -vE "^#|^$"
[DEFAULT]
rpc_backend = rabbit
my_ip = 192.168.142.44
auth_strategy = keystone
use_neutron = True
firewall_driver = nova.virt.firewall.NoopFirewallDriver
debug = True
network_api_class = nova.network.neutronv2.api.API
security_group_api = neutron
linuxnet_interface_driver = nova.network.linux_net.LinuxOVSIfaceDriver
[glance]
api_servers = http://controller1:9292
[keystone_authtoken]
auth_uri = http://controller1:5000
auth_url = http://controller1:35357
memcached_servers = :11211
auth_type = password
project_domain_name = default
user_domain_name = default
project_name = service
username = nova
password = nova
[libvirt]
images_type = rbd
```

① 此功能对 libvirt 和 qemu 版本有要求, 如果在创建虚拟机时出现 libvirt 版本较低的异常, 可以升级 libvirt 或取消此选项。

```

images_rbd_pool = vms
images_rbd_ceph_conf = /etc/ceph/ceph.conf
rbd_user = cinder
rbd_secret_uuid = clb47484-1045-4333-ad7d-55b9354e6f2b
disk_cachemodes="network=writeback"
inject_password = false
inject_key = false
inject_partition = -2
live_migration_flag="VIR_MIGRATE_UNDEFINE_SOURCE,VIR_MIGRATE_PEER2PEER,VIR_MIGRATE_LIVE,
VIR_MIGRATE_PERSIST_DEST,VIR_MIGRATE_TUNNELLED"
hw_disk_discard = unmap
[neutron]
url = http://controller1:9696
auth_url = http://controller1:35357
auth_type = password
project_domain_name = default
user_domain_name = default
region_name = RegionOne
project_name = service
username = neutron
password = neutron
[oslo_concurrency]
lock_path = /var/lib/nova/tmp
[oslo_messaging_amqp]
[oslo_messaging_notifications]
[oslo_messaging_rabbit]
rabbit_host = controller1
rabbit_userid = openstack
rabbit_password = openstack
[vnc]
vnc_enabled = True
vncserver_listen = 0.0.0.0
vncserver_proxyclient_address = 192.168.142.44
novncproxy_base_url = http://192.168.142.41:6080/vnc_auto.html

```

4) 全部计算节点配置完成后，重启每个计算节点的 nova-compute 服务。

```

systemctl restart openstack-nova-compute.service
[root@controller1 ~]# nova service-list

```

```

+-----+-----+-----+-----+-----+-----+
| Id | Binary | Host | Zone | Status | State | Updated_at |
+-----+-----+-----+-----+-----+-----+
| 1 | nova-console | controller1 | internal | enabled | up | 2016-10-15T23:06:41.000000 |
| 2 | nova-conductor | controller1 | internal | enabled | up | 2016-10-15T23:06:41.000000 |
| 4 | nova-scheduler | controller1 | internal | enabled | up | 2016-10-15T23:06:41.000000 |

```

```
| 5 | nova-cert | controller1 | internal | enabled | up | 2016-10-15T23:
06:43.000000 |
| 10 | nova-consoleauth | controller1 | internal | enabled | up | 2016-10-15T23:
06:41.000000 |
| 11 | nova-compute | compute1 | nova | enabled | up | 2016-10-15T23:
06:42.000000 |
| 12 | nova-compute | compute2 | nova | enabled | up | 2016-10-15T23:
06:44.000000 |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
```

10.4.6 Ceph 集成 OpenStack 验证

至此，OpenStack 的 Nova、Cinder 和 Glance 均已成功与 Ceph 集成，现在将 Ceph 作为 OpenStack 的统一后端存储进行验证。验证可按如下步骤进行。

1) 验证镜像是否已经存储在 Ceph 中，如果没有则创建镜像。

在 10.4.4 节中已经创建了基于 Ceph 存储后端的镜像，不妨再次验证镜像已经存储在 Ceph 的 images Pool 中，验证命令如下：

```
[root@controller1 ~]# nova image-list
+-----+-----+-----+-----+-----+-----+
| ID | Name | Status | Server |
+-----+-----+-----+-----+-----+
| d11f5678-9920-4d1a-908f-65df7f942857 | cirros-0.3.4-x86_64 | ACTIVE | |
| 8ca07a37-01c5-4472-be3b-91e0ad0e5c2e | cirros-on-ceph | ACTIVE | |
+-----+-----+-----+-----+-----+-----+
[root@controller1 ~]# rbd ls images
8ca07a37-01c5-4472-be3b-91e0ad0e5c2e
```

2) 创建基于 Ceph 存储后端的 Bootable 存储卷。

对于 Nova 而言，使用 Ceph 提供的 Volume 作为启动盘，则意味着虚拟机运行时镜像文件存放在共享存储上，因此可以很方便地进行 live-migration 操作。这里使用 cirros-on-ceph 镜像创建 Bootable 卷，命令如下：

```
[root@controller1 ~]# cinder create --image-id 8ca07a37-01c5-4472-be3b-91e0ad0e5c2e \
--name ceph-bootable1 --volume-type ceph 2
[root@controller1 ~]# cinder list
+-----+-----+-----+-----+-----+-----+
+-----+-----+
| ID | Status | Name | Size |
+-----+-----+-----+-----+-----+
| Volume Type | Bootable |
+-----+-----+-----+-----+-----+
+-----+-----+
| 13adeafc-7279-4c2b-8e3a-a885ebfd4641 | available | lvm-volume1 | 2 |
| lvm | false |
| 27b33034-950f-415c-98bd-9c462367efdb | available | ceph-bootable1 | 2 |
| ceph | true |
```

```

| 8c6dc44a-6ec3-4187-8de9-5daa1dec0b01 | available | ceph-volume1 | 2 |
| ceph | false |
| ba309a64-29f6-425f-b79a-640c71dbcecf | available | nfs-volume1 | 2 |
| nfs | false |
+-----+-----+-----+-----+
-----+-----+

```

此处必须注意，当 Nova 从 RBD 启动实例时，镜像格式必须是 RAW 格式，否则在启动虚拟机时 glance-api 会报 “TypeError: 'ImageProxy' object is not callable” 错误，并且 Cinder 中会报 “copy image error” 错误。本例使用的 cirros-0.3.4-x86_64-disk.img 镜像并非 RAW 格式，因此需要进行如下转换：

```
qemu-img convert -f qcow2 -O raw cirros-0.3.4-x86_64-disk.img cirros-0.3.4-x86_64-disk.raw
```

转换之后重新进行镜像创建：

```

[root@controller1 ~]# glance image-create --file /data/cirros-0.3.4-x86_64-disk.raw \
--disk-format raw --name "cirros-raw-ceph" --container-format bare --visibility \
public --progress
[root@controller1 ~]# nova image-list

```

```

+-----+-----+-----+-----+
| ID | Name | Status | Server |
+-----+-----+-----+-----+
| d11f5678-9920-4d1a-908f-65df7f942857 | cirros-0.3.4-x86_64 | ACTIVE | |
| 8ca07a37-01c5-4472-be3b-91e0ad0e5c2e | cirros-on-ceph | ACTIVE | |
| f6324b23-4249-4067-a885-b44cde7b758a | cirros-raw-ceph | ACTIVE | |
+-----+-----+-----+-----+

```

使用新镜像创建新的 Bootable 卷 ceph-bootable2：

```

[root@controller1 ~]# cinder create --image-id f6324b23-4249-4067-a885-b44cde7b758a \
--name ceph-bootable2 --volume-type ceph 2
[root@controller1 ~]# cinder list

```

```

+-----+-----+-----+-----+
| ID | Status | Name | Size |
+-----+-----+-----+-----+
| 13adeafc-7279-4c2b-8e3a-a885ebfd4641 | available | lvm-volume1 | 2 |
| lvm | false |
| 27b33034-950f-415c-98bd-9c462367efdb | available | ceph-bootable1 | 2 |
| ceph | true |
| 8c6dc44a-6ec3-4187-8de9-5daa1dec0b01 | available | ceph-volume1 | 2 |
| ceph | false |
| ba309a64-29f6-425f-b79a-640c71dbcecf | available | nfs-volume1 | 2 |
| nfs | false |
| c085140d-0f8f-43e7-a77d-711a80293732 | available | ceph-bootable2 | 2 |
| ceph | true |

```

3) 从基于 Ceph 存储后端的 Volume 启动虚拟机。

```
[root@controller1 ~]# nova boot --flavor 1 --boot-volume c085140d-0f8f-43e7-a77d-711a80293732 \
--key-name admin-key --security-group default --nic net-id=1d6096e3-f042-434b-98fe-2e4e5425d44d \
```

```
ceph-instance1
```

```
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State	Networks
...4b169b5491	ceph-instance1	ACTIVE	-	Running	admin-net1=192.128.1.8

因为虚拟机是从 Volume 启动的，所以此时 Ceph 的 vms Pool 中应该没有虚拟机镜像，虚拟机镜像位于 volumes Pool 中，即虚拟机运行在 Ceph 的 volumes 这个 Pool 中：

//输出为空

```
[root@controller1 ~]# rbd ls vms
```

```
[root@controller1 ~]#
```

4) 从 Ceph RBD 直接启动虚拟机。

```
[root@controller1 ~]# nova boot --flavor m1.tiny --image cirros-raw-ceph --nic
net-id=67fb0d01-28fd-4cb4-871b-c1517561f192 --security-group default --key-
name admin-key ceph-instance2
```

```
[root@controller1 ~]# nova list
```

ID	Name	Status	Task State	Power State	Networks
...4b169b5491	ceph-instance1	ACTIVE	-	Running	admin-net1=192.128.1.8
...235925892e	ceph-instance2	ACTIVE	-	Running	admin-net2=192.128.2.10

此时 Ceph 的 vms 这个 Pool 中应该出现虚拟机镜像，即此时的 VM 已经直接从 Ceph 的 RBD 启动，Ceph RBD 成为 Nova 的临时后端存储：

```
[root@controller1 ~]# rbd ls vms
```

```
f0717c2e-3765-4964-9244-be235925892e_disk
```

5) 对 RBD 启动的虚拟机 ceph-instance2 进行 live-migration。

//live-migration前，ceph-instance2位于compute2节点

```
[root@controller1 ~]# nova hypervisor-servers compute2
```

ID	Name	Hypervisor ID
Hypervisor Hostname		


```

-----+
| f0717c2e-3765-4964-9244-be235925892e | instance-0000002e | 1 |
compute2 |
-----+-----+-----+-----+-----+
//开始live-migration迁移
[root@controller1 ~]# nova live-migration ceph-instance2 compute1
[root@controller1 ~]# nova list
-----+-----+-----+-----+-----+-----+
| ID | Name | Status | Task State | Power State | Networks |
-----+-----+-----+-----+-----+-----+
| ...169b5491 | ceph-instance1 | ACTIVE | - | Running | admin-net1=192.128.1.8 |
| ...5925892e | ceph-instance2 | MIGRATING | migrating | Running | admin-net2=192.128.2.10 |
-----+-----+-----+-----+-----+-----+
[root@controller1 ~]# nova list
-----+-----+-----+-----+-----+-----+
| ID | Name | Status | Task State | Power State | Networks |
-----+-----+-----+-----+-----+-----+
| ...169b5491 | ceph-instance1 | ACTIVE | - | Running | admin-net1=192.128.1.8 |
| ...5925892e | ceph-instance2 | ACTIVE | - | Running | admin-net2=192.128.2.10 |
-----+-----+-----+-----+-----+-----+
//迁移后两个虚拟机全部位于compute1节点上
[root@controller1 ~]# nova hypervisor-servers compute1
-----+-----+-----+-----+-----+-----+
| ID | Name | Hypervisor ID | Hypervisor |
-----+-----+-----+-----+-----+-----+
| Hostname |
-----+-----+-----+-----+-----+-----+
| e228988e-6d24-4964-be13-0e4b169b5491 | instance-0000002b | 1 | compute1 |
| f0717c2e-3765-4964-9244-be235925892e | instance-0000002e | 1 | compute1 |
-----+-----+-----+-----+-----+-----+
-----+

```

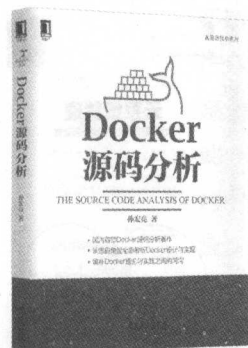
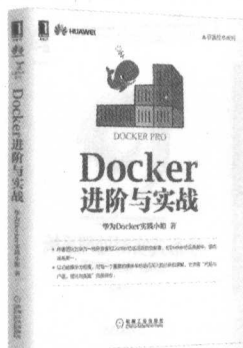
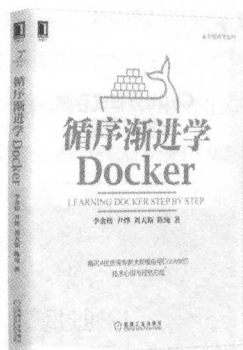
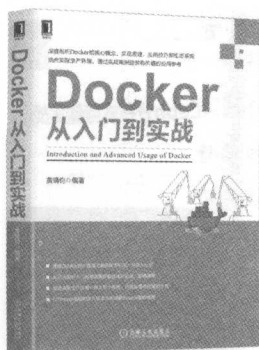
本节的验证充分说明，OpenStack 的 Glance、Cinder 和 Nova、Ceph 集成后，OpenStack 原生各项功能均可正常使用。此外，当 Ceph RBD 作为 Nova 的临时后端存储时，即使虚拟机不从 Volume 启动，也可以进行 live-migration 操作。对于 Nova 而言，Ceph RBD 等效于共享存储，因此 Nova 虚拟机可以很轻松地实现 live-migration。

10.5 本章小结

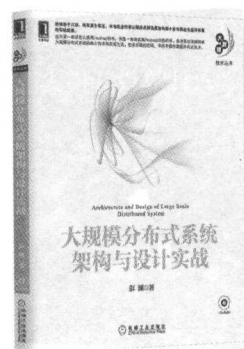
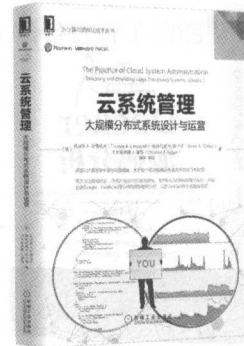
数据存储服务是所有 IaaS 架构都必须考虑的问题。在 OpenStack 中，数据存储服务分为临时性存储和持久性存储。临时性存储主要用于虚拟机的临时系统盘，而持久性存储主要用于存储用户数据。临时性存储通常在虚拟机生命结束时便随之消失，而持久性存储即使在虚拟机被终止后其上存储的数据仍然可被再次使用。OpenStack 中持久性存储主要分为对象存储、块存储和文件系统存储，其中对象存储由 Swift 提供，而块存储由 Cinder 项

目提供。从 OpenStack 社区的部署情况来看，Cinder 块存储是使用最为普遍的存储方式，本章对 Cinder 块存储从架构和工作原理方面进行了介绍，并通过 LVM 和 NFS 插件实现了 Cinder 的多后端存储方案。用户在 OpenStack 中创建云盘时，只需指定云盘类型即可使用基于不同存储后端实现的云盘。除了 Cinder 和 Swift 之外，在 OpenStack 中呼声很高并被普遍使用的存储方案便是 Ceph，Ceph 作为一种统一、分布式存储集群，能够向客户端同时提供对象存储、块存储和文件系统存储服务，并且 Ceph 实现了“无须查表，算算即可”的数据访问方式，同时通过 CRUSH 算法实现了数据在各个存储节点近似均匀地动态分布和自我愈合的能力。Ceph 存储集群还具有无限可扩展、高性能和高可用等特性。目前 OpenStack 中的 Nova、Cinder、Swift 和 Glance 项目均实现了对 Ceph 存储集群的支持，通过与 Ceph 存储集群的集成，OpenStack 的这些项目均可使用 Ceph 作为其数据存储后端，从而将 OpenStack 各个项目分散存储的数据统一存储到 Ceph 存储集群中，实现了数据与应用程序的分离，并通过统一存储的方式便于数据的维护和管理。本章对 Ceph 存储集群的起源、背景、内部架构和实现原理均作了详细介绍，并以实战的形式讲解了如何部署 Ceph 存储集群，以及如何将 OpenStack 的 Nova、Cinder 和 Glance 项目与 Ceph 进行集成，并在最后对集成 Ceph 后的各个 OpenStack 项目进行了功能验证。

推荐阅读



推荐阅读



作者简介

山金孝（Warrior）国内较早接触OpenStack的一线技术专家，长期致力于OpenStack的研究、实践和生产环境部署，是OpenStack社区的积极参与者和实践者。作为由传统IT架构转型为云计算领域的技术专家，参与并设计实施了移动、电信、联通、招行、国家电网和长安汽车等多家大中型企业的高可用业务系统，在系统容灾和高可用集群建设上具有多年的项目实施经验。

曾就职于IBM，现就职于招商银行，主持设计并实施了招商银行重庆分行的OpenStack高可用生产系统集群，目前是招商银行重庆分行核心业务系统和云计算基础架构平台的主要负责人。

此外，他还是IBM认证的高级技术专家和DB2方向的高级DBA，同时也是RedHat认证的Linux系统工程师。

OpenStack Cluster HA

Principles and Architecture

本书是对OpenStack高可用集群部署和实现的多维深度实践，总结了OpenStack高可用的不同方案，并详细讲解了计算、存储和网络各个模块的高可用架构及实施。难能可贵的是，本书没有停留在理论和实验环境层面，而是总结了大量生产环境的实践。

—— 肖力 云技术社区创始人

金孝具有多年金融行业及大型制造业的云计算从业经验，经历过诸多大中型企业的核心系统项目建设，在云计算及虚拟化方面积累了丰富的项目经验，也是国内较早一批接触OpenStack并对其进行研究和部署实践的开拓者。这是一本真正由OpenStack终端用户编写，并且面向生产环境部署的专著，书中有大量代码和实施步骤，相信对OpenStack的落地和运维能够起到积极的推动作用。

—— 张鹏 IBM全球技术服务部高级工程师/客户服务经理

长久以来，一直期望有一本全方位讲解OpenStack高可用部署与实施的图书，让更多的工程师能够理解、掌握和实施面向生产系统的OpenStack云计算项目。很欣慰能够看到本书的面世，它从理论到实战部署，再到运维，全方位讲解了OpenStack的高可用集群。

—— 刁坤华 重庆奇梦达科技有限公司创始人

作者具有多年OpenStack的项目实施经验，本书采用理论与实践相结合的方式，由浅入深地讲解了在生产系统中部署OpenStack高可用集群的方法，总结了实践中常见问题的解决方案，是为即将和正在使用OpenStack的云计算工程师准备的“核武器”。

—— 宋珩 招商银行重庆分行信息技术部总经理

金孝具备深厚的理论功底和丰富的行业实战经验，不同于一般作者，他开展了非常多的系统性的工程实践，积累了丰富的实战经验。他一直致力于研究最新的云计算技术，始终奋斗在最前沿，做出了很多卓有成效的探索和实践。本书理论与实践相结合，非常适合OpenStack初学者、架构师、运维工程师等人员阅读。相信您从本书中一定能有宝贵的收获。

—— 周鹏 招商银行信息技术部高级工程师



投稿热线: (010) 88379604

客服热线: (010) 88379426 88361066

购书热线: (010) 68326294 88379649 68951800

官方网站: www.hzbook.com

网上购书: www.china-pub.com

数字阅读: www.hzmedia.com.cn

上架指导: 计算机/云计算

ISBN 978-7-111-57570-2



9 787111 575702 >

定价: 99.00元